



Learning jQuery Deferreds

TAMING CALLBACK HELL WITH DEFERREDS AND PROMISES

Terry Jones & Nicholas H. Tollervey

Learning jQuery Deferreds

Orchestrating asynchronous function calls in JavaScript often leads to callback hell, but there is a reliable way to avoid this painful state of affairs. With this concise and simple guide, you'll learn how to use jQuery deferreds and promises, an elegant approach for managing asynchronous calls in both client and server applications.

This book contains 18 examples that use deferreds to solve progressively challenging real-world programming problems, along with 75 stimulating puzzles (and their solutions) that will help you understand how and when to use deferreds. You'll learn new tricks in a fun way, and become immersed in the practice of event-based programming.

- Understand the logic behind creating deferreds and returning promises
- Get a structured explanation of jQuery's deferred API
- Delve into the dynamics of using deferreds
- Explore a broad collection of useful deferred recipes developed by the authors
- Gain hands-on experience by solving challenges that accompany each recipe
- Go deeper into deferreds: encounter novel abstractions and mind-bending use cases

Terry Jones is currently CTO of Fluidinfo, which he founded in 2007. He also works on virus discovery in the Zoology department at the University of Cambridge.

Nicholas H. Tollervey is a classically trained musician, philosophy graduate, teacher, writer, and software developer. He's just like this biography: concise, honest, and full of useful information.

US \$29.99

CAN \$31.99

ISBN: 978-1-449-36939-2



5 2 9 9 9

Twitter: @oreillymedia
facebook.com/oreilly

Learning jQuery Deferreds

Terry Jones and Nicholas H. Tollervey

Learning jQuery Deferreds

by Terry Jones and Nicholas H. Tollervey

Copyright © 2014 Terry Jones and Nicholas H. Tollervey. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://my.safaribooksonline.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editors: Simon St. Laurent and Meghan Blanchette

Proofreader: Kristen Brown

Production Editor: Kristen Brown

Cover Designer: Karen Montgomery

Copyeditor: Charles Roumeliotis

Interior Designer: David Futato

January 2014: First Edition

Revision History for the First Edition:

2013-12-20: First release

See <http://oreilly.com/catalog/errata.csp?isbn=9781449369392> for release details.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Learning jQuery Deferreds*, the image of a musky rat-kangaroo, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-449-36939-2

[LSI]

Table of Contents

Preface.....	vii
1. Introduction.....	1
Food for Thought	1
Terminology: Deferreds and Promises	3
Familiar Promises	4
2. The jQuery Deferred API.....	7
Consuming Promises	8
More Terminology: Resolve, Reject and Progress	8
done	9
fail	9
always	10
progress	10
promise	11
then	11
state	13
when	13
Creating Deferreds	15
Construction	15
resolve and resolveWith	16
reject and rejectWith	17
notify and notifyWith	17
Putting It All Together	17
Deferred Dynamics	18
Deprecated Promise Methods	19
isRejected and isResolved	19
pipe	19

Changes in the jQuery Deferred API	19
3. Deferred Recipes.....	21
A Replacement for the setTimeout Function	21
Challenges	22
Messaging in Chrome Extensions	23
Challenges	24
Accessing Chrome Local Storage	25
Challenges	26
Running Promise-Returning Functions One by One	26
Challenges	28
A Promise Pool with an emptyPromise Method	28
Creating a Promise Pool	29
Using the Promise Pool	30
Challenges	31
Displaying Google Maps	32
Challenges	36
Communicating with a Web Worker	37
The Web Worker Code	37
Creating a Web Worker	39
Using It	40
Summary	41
Challenges	41
Using Web Sockets	42
The Web Socket Server	42
The Web Socket Client	44
Challenges	46
Automatically Retrying Failing Deferred Calls	47
Challenges	48
Memoization	49
Discussion	50
Avoiding the Dogpile Effect	51
Challenges	51
Short-Term Memoization of In-Progress Function Calls	52
createUser Is Not Idempotent	53
Challenges	53
Streaming Promise Events	54
Delivering More Information	54
Delegating the Event Stream	55
To Be Continued...	56
Challenges	56
Getting the First Result from a Set of Promises	57

Which Promise Fired?	58
A Fly in the Soup	59
delegateEventStream Redux	59
Challenges	60
A Deferred Queue	61
Challenges	63
when2: An Improved jQuery.when	63
Using when2 to Time Out a Single Promise	67
Differences from \$.when	68
Challenges	69
Timing Out Promises	69
Challenges	72
Controlling Your Own Destiny	73
Challenges	74
Deactivating a Promise	74
Challenges	76
4. More Time in the Mental Gymnasium.....	77
Do You Really Understand jQuery Deferreds?	77
Promises/A+	78
Promises Are First-Class Objects for Function Calls	79
Asynchronous Data Structures	80
Advantages of Deferreds	81
Difficulties with Deferreds	82
Further Reading	83
A. Hints for Selected Challenges.....	85
B. The Promises/A+ Specification.....	107
C. Converting an ArrayBuffer to Base 64.....	113

Preface

The world of JavaScript has changed significantly in recent years with more sophistication in client-side JavaScript applications and the arrival of server-side JavaScript using `node.js`.

In building increasingly complex applications, JavaScript programmers have had to become more adept at dealing with asynchronous APIs. In earlier years, JavaScript was all client side. Programmers were only required to deal with single, independent asynchronous function calls, whose results were used to update an HTML user interface.

The situation today is far richer. Server-side JavaScript applications regularly make multiple asynchronous calls to many other services to produce responses: to databases, caches, load balancers, the filesystem, authentication systems, third-party APIs, etc. Meanwhile, client-side JavaScript now has routine access to dozens of asynchronous APIs, such as those provided by HTML5 as well as good old AJAX (remember, the first A in AJAX stands for asynchronous). Applications need to be able to coordinate simultaneous calls to multiple asynchronous APIs: to get the fastest result, to combine information, to wait for multiple calls to complete, to execute calls in specific orders, to alter the flow of control depending on results, to deal with errors, to fall back to alternate services (e.g., on cache misses), to retry failing network calls, and so on.

“Callback hell,” a phrase with about 10,000 Google hits, describes what happens when programmers try to build modern applications with old-school single-callback programming. For many programmers, this pain is their daily reality. Using single callbacks to build applications that need to do even basic coordination of asynchronous calls can be very difficult. You have to think hard and often end up with complicated, brittle, and opaque solutions. These contain hard-to-find bugs, are hard to document, and can be very hard to extend when additional asynchronous requirements enter the picture. Coding for multiple asynchronous events with the single-callback model is a real challenge, even for very smart and experienced programmers.

About You

Firstly, we're writing for jQuery programmers who do not know about deferreds. We've found that most programmers who use jQuery have never heard of deferreds. Among those who have, there are many who find deferreds confusing or who are under the false impression that they are too abstract or difficult to understand. Deferreds are a misunderstood yet powerful programming paradigm. Recently, deferreds have been added to many JavaScript libraries and frameworks, and are now attracting widespread attention. A [search for "deferreds" on StackOverflow](#) currently gives over 18,000 hits, up 40% in the six months since we started this book.

We also want to help JavaScript programmers, *both client side and server side*, who know about deferreds but aren't making heavy use of them. If you'd like to beef up your practical knowledge, to see more examples in action, and to think about deferreds from different angles, we'd love to have you as a reader. We want to help you stretch your mind, in both breadth and depth; the book has 18 real-world examples of deferred use along with 75 challenges (and their solutions) to push your thinking.

Finally, and most ambitiously, we hope we've written a book that will be useful and stimulating to programmers using deferreds and promises beyond those in jQuery and even beyond JavaScript. The conceptual underpinnings of deferreds are almost identical across the many JavaScript packages and other programming languages that support them. Because the concepts are so similar and so few, you'll find it straightforward to port code between implementations. Virtually everything you learn in this book will be useful for working with other flavors of deferreds. We want it to be a fun and valuable read, no matter what your preferred language is. We've tried to write the book we wish had been available as we learned deferreds ourselves.

Our Aims

In this book we'll teach you how to avoid callback hell by using deferreds.

But there's *much* more to deferreds than that. Deferreds provide something that was not there before: a simple mechanism for dealing with future results. This gives you the opportunity to do things in different ways that go beyond mere simplification of syntax. It gives you the opportunity to really *think* about the practice of programming and to broaden your mental toolkit. The thinking can of course be critical, including conclusions about which deferred package to use or whether it is even sensible to use deferreds in a given situation. For us, the process of learning about and beginning to appreciate programming with deferreds was a feeling of our brains growing new muscles.

Our primary aim is to introduce deferreds to programmers who have had no exposure to them. We're aiming to give you a really strong *general and concrete* understanding of what deferreds are and how to wield them. If we succeed, you'll be fully confident

when encountering deferreds in any other context: whether with another JavaScript deferred package or in a different programming language.

A secondary aim is to provide a broad collection of examples of nontrivial real-world deferred uses. We’ve been programming with deferreds (in Python and JavaScript) for the last 7 years and have pulled together some of the most useful examples we’ve built in that time. These are usually relatively short snippets of code, around 100 lines, but sometimes require careful thought to develop. We hope the detailed recipes in [Chapter 3](#) will be a place you’ll return to for ideas on how to approach deferred problems you face.

Challenges

The deferred recipes in [Chapter 3](#) all leave you with a set of challenges. These are meant to encourage you to engage with and think more deeply about the material just presented. If you want to write code as well, that’s a bonus. The main point, however, is to *think*. Don’t be a passive reader! Working with deferreds very often requires focused thinking about how problems, and small variations on them, might be solved. Once you “get” deferreds, solving puzzles with them can be very engaging. The more you do it, the better you get, and the more you see their flexibility and power. Above all though, figuring out how to do things with deferreds is just plain fun!

If you’re stuck on a challenge, you’ll find hints and solutions in [Appendix A](#).

jQuery Deferreds

We chose to focus on jQuery deferreds because jQuery is ubiquitous and because two important and familiar aspects of jQuery (`$.ajax` and `animations`) already have deferred support built in.

However, jQuery deferreds are certainly not the last word on the subject. As you will see, they differ markedly in one important respect from the many (at least 35) other deferred packages for JavaScript: jQuery deferreds do not currently follow the excellent [Promises/A+](#) specification (see [“Promises/A+” on page 78](#)).

This book was written for the [1.10.2](#) or [2.0.3](#) versions of [jQuery](#). Because the jQuery deferred API has changed several times, it is important to know where you stand, as we’ll see in [“Changes in the jQuery Deferred API” on page 19](#). Server side, we’re using version 1.9.1 of the [jQuery-deferred node module](#). The node module and much of the jQuery deferred code was written by [Julian Aubourg](#).

Our JavaScript Coding Style, or Lack Thereof

JavaScript is not a “there’s only one way to do it” language. As a result, almost every line of code in the book could have been written differently. To keep the focus on deferreds, we’ve chosen to write code in the simplest/clearest possible way. It will be up to you to slightly change our examples to fit whatever coding style or framework you’re using to, for example, create JavaScript objects (via `new`, using self-calling anonymous functions, global variables, etc.), loop (for loop, `$.map`, `[]`.`forEach` etc.), write variable declarations, log via `console.log`, and so on. Also, in the name of keeping the focus on deferreds, we often ignore (or almost ignore) error processing.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.



This icon signifies a general note.



This icon indicates a warning or caution.

Using Code Examples


Supplemental material (code examples, exercises, etc.) is available for download at <https://github.com/jquery-deferreds/code>.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. Attribution usually includes the title, author, publisher, and ISBN. For example: “*Learning jQuery Deferreds*, by Terry Jones and Nicholas H. Tollrvey (O'Reilly). Copyright 2014 Terry Jones and Nicholas H. Tollrvey, 978-1-4493-6939-2.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

Safari® Books Online

 *Safari Books Online* is an on-demand digital library that delivers expert **content** in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of **product mixes** and pricing programs for **organizations**, **government agencies**, and **individuals**. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers including O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens **more**. For more information about Safari Books Online, please visit us **online**.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax).

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <http://oreil.ly/learn-jquery-deferreds>.

To comment or ask technical questions about this book, send email to bookquestions@oreilly.com.

For more information about our books, courses, conferences, and news, see our website at <http://www.oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

Acknowledgments

Thanks to the Python **Twisted community**, who have helped with our ongoing deferred education, and whose thinking has enormously influenced the design of deferreds in other languages.

Thanks to **Fluidinfo** for the various open source (Twisted) deferred code it has published, and for the opportunity to work with and learn about deferreds in depth.

Thanks to Justin Wohlstadter of Wayfinder for allowing us to adapt some Coffeescript examples one of us wrote for him.

Thanks to the jQuery developers and especially to Julian Aubourg for adding deferreds to jQuery and for extracting that code to produce the **jQuery-deferred node module**.

Thanks to Francesco Agati, Michael Chermiside, Jonathan Dobson, Tim Golden, Peter Inglesby, Robert Rees, David Semeria, and Justin Wohlstadter for their careful and useful reviews.

Thanks to the professional and efficient editing and production team at O'Reilly: Meghan Blanchette, Kristen Brown, Charles Roumeliotis, and Simon St. Laurent.

Terry would like to thank Ana, Sofia, Lucas, Findus, and the **Flying Spaghetti Monster**.

Nicholas would like to thank Mary, Penelope, Sam, and William for their continued support and leg-pulling.

CHAPTER 1

Introduction

A deferred represents a result that may not be available yet. It is an abstraction for something that has yet to be realized.

We attach code to the deferred to take care of the expected or erroneous result when it becomes available.

That's it!

Deferred usage is very similar to the way we make plans: when *X* finishes, if there was no error, do *Y*, otherwise do *Z*. To give an everyday example, “when the tumble dryer finishes, if the clothes are dry, fold them and put them away, otherwise hang them on the line.” Here, “the tumble dryer finishes” is the deferred, “fold them” and “put them away” are handlers for the good case (also known as *callbacks*) and “hang them on the line” is the handler for an error condition (sometimes known as an *errback*). Once the plan is made, we're free to get on with something else.

Although the outcome of the deferred is undetermined, we can plan ahead for two possibilities: the clothes are either going to be wet or dry when the tumble dryer finishes. The dryer may actually already be finished, but that does not impact our planning. The important thing to note is that deferreds provide a clear separation between initiating something (resulting in a deferred) and handling the result (when the deferred completes). There are many advantages from this clean separation, and in this book we'll explore them.

Don't panic if this all seems a bit abstract; there are plenty of examples coming right up.

Food for Thought

JavaScript programs operate in an **event-based environment**. Let's be clear about what that means. Keystrokes, mouse clicks, and low-level I/O operations completing are all events. As your program runs, it can, in advance, tell the JavaScript runtime about events

it is interested in and provide code to execute when such events occur. Later, when events relevant to your program happen, the JavaScript runtime will invoke the code you wrote to handle them.

This is a simple, efficient, and familiar model. It closely matches the way we plan ahead in our daily lives. Most of us could quickly make a breakfast of fresh orange juice, toast, and boiled eggs by preparing all three items at once. We know that we'll have time to squeeze the oranges *while* the toast and the eggs are cooking, so we'll get them both cooking first. We know what to do, regardless of whether the toast or the eggs are cooked first. On a grander scale, consider the kitchen staff of a busy restaurant. By initiating long-term actions (e.g., putting water on to boil), by reacting to events (e.g., the cheese on a dish is browning), and by switching among other tasks in the meantime, a small team can efficiently prepare a wide range of dishes for a large number of simultaneous diners.

In event-based programming (and not only in JavaScript), handling single events is trivial. Coordinating code to handle multiple events, though, can be very challenging. Ad hoc solutions are often awkward to construct, difficult to test, brittle, hard for others to follow, and depressing to maintain.

The problem rears its head even in trivial situations. For example, suppose you have a JavaScript food API available, with `makeToast` and `makeEggs` functions. Both accept a callback function that they call once their product is done, passing the finished result as an argument. An example call looks like:

```
makeToast(function(toast){  
    // Toast is ready!  
});
```

Your challenge is to write a function called `makeBreakfast` that gets the toast and the eggs cooking simultaneously and that calls a single callback when both are ready.

```
function makeBreakfast(callback){  
    // Use makeToast and makeEggs to make toast and eggs simultaneously.  
    // When both are ready, pass them to callback.  
}
```

Pause now, please, and think about how you'd implement `makeBreakfast`.

Here's a common strategy: when either underlying function (`makeToast` or `makeEggs`) finishes, check to see if the other result is also available. If so, call the callback. If not, store the result so the termination of the other function can pass it to the callback. The resulting code isn't elegant and doesn't generalize well. What if making breakfast expands to also include making coffee and pancakes?¹

1. See “when” on page 13 for jQuery's solution.

This is an extremely trivial example of managing multiple events, yet our code is already a mess. Real-world scenarios are almost always more complex, and can of course be *much* more complex.

If you had to solve problems like the above a few times, you'd soon see a general pattern. You'd likely write a helper function or two. And if you did that, *you'd be well on your way to implementing deferreds!*

Terminology: Deferreds and Promises

We need to get a little terminology clear from the very beginning: the difference between *deferreds* and *promises* in jQuery.²

Continuing with our food theme, the first edition³ of *Twisted Network Programming Essentials* by Abe Fettig (O'Reilly) gives a beautiful analogy of deferreds in the real world. Some popular restaurants use remotely activated buzzers to let diners know when a table is available. This avoids a physical queue of waiting customers clogging up the entrance to the restaurant and allows future diners to enjoy a drink at the bar or a short walk in the interim. This elegant approach moves us from a problematic and boring synchronous solution (waiting in line) to an asynchronous one that lets everyone get on with other things in the meantime.

When the maître d'hôtel puts your details (number of diners, seating preference, etc.) into the restaurant's system, he or she is taking the first in a series of steps that will result in you eventually getting a table. In jQuery terminology, the maître d' creates a *deferred*. You are handed a buzzer, which corresponds to a *promise*. At some point in the future, when a table becomes free, the maître d' will push a button or click a mouse to "resolve" the deferred and the buzzer will go off in your pocket. Importantly, *you* (the holder of the promise), cannot cause the buzzer to go off. Only the maître d' (the holder of the deferred) can do that.

With jQuery deferreds, things work in exactly the same way. The programmer writing a function that needs to get some slow work done (for example, a database operation or a network call) creates a deferred and arranges to fire it when the result of the work becomes available. From the deferred a promise is obtained and returned to the caller. Just like the diner with the buzzer, the caller cannot cause the promise to fire. Just as future diners can have a drink at the bar, a program that receives a promise can get on

2. Note that jQuery's terminology (and implementation) is slightly different from other packages, some of which do not use the term "deferred" at all. At some point you might like to read the Wikipedia article on "[Futures and promises](#)".

3. Jessica McKellar was added as an author in the second edition and the nice analogy was removed. Our reference is to the second edition.

with other computations instead of twiddling its thumbs while waiting for the promise to fire.

To summarize: create deferreds but return promises.

Familiar Promises

If you’ve ever used `$.ajax` in jQuery or used any of the `animate` methods, you’ve already used a promise. For example, you may have written:

```
$('#label').animate({ opacity: 0.25 }, 100, function(){  
    // Animation done.  
});
```

The return value of the `animate` function gives you a way to get a promise that is resolved when the animation finishes. You could instead have written:

```
var promise = $('#label').animate({ opacity: 0.25 }, 100).promise();  
  
promise.done(function(){  
    // Animation done.  
});
```

That may not seem like a big deal, but what if you want to coordinate what happens after two animations have finished? Using promises, it’s trivial:

```
var promise1 = $('#label-1').animate({ opacity: 0.25 }, 100).promise();  
var promise2 = $('#label-2').animate({ opacity: 0.75 }, 200).promise();  
  
$.when(promise1, promise2).done(function(){  
    // Both animations are done.  
});
```

The jQuery `$.when` method can accept multiple promises and return a new one that will let you know when all the passed promises have resolved. Contrast the simplicity of the above with the breakfast-making shenanigans in “Food for Thought” on page 1.

The `$.ajax` method returns a value that also has promise methods. So, you could write the following:

```
$.when($.ajax('http://google.com'), $.ajax('http://yahoo.com')).then(  
    function(googlePage, yahooPage){  
        // Both URLs have been fetched.  
    }  
);
```

It’s easy to do more complex things, e.g., fetch the contents of two URLs, run an animation after each loads, and then do something else when all four events are finished:

```
$.when(  
    $.ajax('http://google.com').then(function(){  
        return $('#label_1').animate({ opacity: 0.25 }, 100);  
    })  
);
```

```

    }},
    $.ajax('http://yahoo.com').then(function(){
        return $('#label_2').animate({ opacity: 0.75 }, 200);
    })
).then(
    function(){
        // Both URLs have been fetched and both animations have completed.
    }
);

```

Notice how the code almost reads like a natural language description of a simple plan.

Unfortunately, deferreds have a reputation for being abstract and difficult to understand. As we've seen though, they're not! They're conceptually very close to the way we naturally think about and plan for future events.

The basic understanding of deferreds provided in this chapter is all you need to enjoy some of the mind-bending, elegant, and downright fun ways in which deferreds can make event-based programming so challenging and rewarding. We'll see a ton of examples of using deferreds in [Chapter 3](#). But before we do that, we'll need to learn about the jQuery deferred API.

The jQuery Deferred API

There are different levels at which you can learn about jQuery deferreds, and these each give a different perspective.

At the lowest level there is the JavaScript source, the jQuery *deferred.js* and *call-backs.js* files. Reading the source is very informative, but it's definitely not the simplest JavaScript to understand! Besides being challenging to follow (jQuery is optimized for code size and execution speed, not readability), the source also doesn't tell you what deferreds are for or how to use them. From reading the source, it's not even clear what the methods available on deferreds might be.

Next, there's the official jQuery documentation for the **Deferred object**, `jQuery.when` (which we'll refer to as `$.when` from now on), and the `.promise()` function for DOM element collections. The API documentation tells you what methods are available, what their arguments are, methods that are deprecated or that have changed between versions, etc. You'll want to read the official documentation closely and will probably return to it many times as you become increasingly fluent with deferreds.

A further level is a proposal (see **"Promises/A+" on page 78**) for standardizing the behavior of promises across JavaScript libraries. While not directly associated with jQuery's deferreds and promises, it illustrates the guidance that informed the implementation of the API.

What's missing is a higher-level discussion that explains the API and the dynamics of deferreds. That's what we aim to provide in this chapter. We discuss every API method and often show examples of API calls, but we're not attempting to duplicate the online documentation. Read our description to understand the API and then consult the official documentation if you need more detail.

A natural way to begin is to first look at what you can do when a promise is returned by a function you call. Once you understand that, it's easy to learn how to make deferreds yourself so that your code can return promises to others.

Consuming Promises

What kinds of functions return promises?

There are three ways you can receive a promise using jQuery (two of which we mentioned in “Familiar Promises” on page 4). First, `$.ajax` returns a promise.¹ Most jQuery users are used to using `$.ajax` by passing error and success functions in the call to create the request. But you can also treat the return value of `$.ajax` as a promise and reap the benefits of operating on deferreds. Second, we’ve seen that jQuery `animate` methods return an object with a `promise` method that returns you a promise. Third, when you select a set of DOM elements (e.g., `$('p')` to select all HTML `<p>` elements), the result is an object with a `promise` method that also returns a promise. By default it resolves when all animations on the selected elements are finished.

In addition to these, you might receive promises from function calls to other (non-jQuery) JavaScript APIs.

However it happens, if you have your hands on a promise, it’s important to understand what you can do with it.

More Terminology: Resolve, Reject and Progress

The first thing to understand about a promise is that it was created from a deferred. Whoever made the deferred is going to arrange for the promise they gave you to deliver you a value. jQuery uses *resolve*, *reject*, and *progress* to describe the things that can happen to your promise. If nothing goes wrong, the promise will be resolved. If an error occurs, it will be rejected. Along the way, the deferred might make measurable progress and report this to your promise. In this book we’ll sometimes informally say a deferred or a promise has *fired*. By this we mean that the deferred was rejected or resolved, but that we don’t care which.

1. Actually, this is not strictly true. `$.ajax` returns an object that has methods that point to the promise methods on a deferred that `$.ajax` uses internally. For regular jQuery users, this detail is of no importance and can be ignored.



In the following API examples, we assume you have a variable called `promise` obtained from a function that created a deferred and returned its promise.

done

Use the **done method** on a promise to arrange for a function to be called if the deferred is resolved:

```
promise.done(function(result){  
    // result is the value with which the deferred was resolved.  
    console.log('The promise was resolved with', result);  
});
```

`done` can be called many times on the same promise. Each call results in the passed function being added to a list of functions that will be called when the promise is resolved.

The function you pass to `done` will be called with all the arguments the deferred is resolved with. The above example just shows the simple case of a deferred being resolved with one argument.

Note that there is no point in returning a value from a `done` callback! Any returned value will simply disappear. It will not be passed on. It will not be given to other `done` callback functions. All `done` callbacks are independent. *They will all be called with the same value.* If you want to modify the result of a promise so as to pass the modified value along, you'll need to use `then`, explained [on page 11](#).

fail

Use the **fail method** on a promise to arrange for a function to be called if the deferred is rejected:

```
promise.fail(function(error){  
    // error is the value with which the deferred was rejected.  
    console.log('The promise was rejected with', error);  
});
```

As with `done`, `fail` can be called multiple times to add failure functions. Functions passed to `fail` will be called with the full set of arguments the deferred was rejected with. Also, as with callbacks attached via `done`, there is no point in returning a value from a `fail` callback.

always

If you want a function to be called no matter whether the original deferred is resolved or rejected, you can pass it to the **always method** on the promise. This is useful, for example, in cleaning up or logging:

```
promise.always(function(value){  
    // The deferred fired with value, either via its resolve or reject method.  
    console.log('The promise fired with value', value);  
});
```

Note that in the callback function above, you may not be able to tell whether the value comes from the deferred being resolved or rejected, and *you shouldn't care*. If you do care, you should probably be using **done** and/or **fail**.

As with **done** and **fail**, **always** can be called multiple times to add functions that should always be called. All arguments used to reject or resolve the deferred will be passed along to the promise **always** functions. Also, as with callbacks attached via **done** and **fail**, there is no point in returning a value from an **always** callback.

progress

Deferreds may be able to report periodically on the progress of the asynchronous activity they're associated with. A **progress callback** can be used to handle these events:

```
promise.progress(function(value){  
    // A progress value has been reported by the deferred.  
    console.log('Progress!', value);  
});
```

Note that a **progress** function can be called many times. This can be used to support, for example, a user interface that updates a progress bar as an operation proceeds.

jQuery calls **progress** callbacks every **jQuery.fx.interval** milliseconds on elements that are being animated. See the entry for the **progress** attribute of the **jQuery animate documentation**. In reading the **animate** documentation, note that passing a **progress** function to the **animate** constructor via:

```
$('#something').animate({..., progress: function(){ ... }})
```

is equivalent to:

```
$('#something').animate(...).promise().progress(function(){ ... })
```

As with **done**, **fail**, and **always**, **progress** can be called multiple times to add functions that should receive progress information. Also, as with callbacks attached via **done**, **fail**, and **always**, there is no point in returning a value from a **progress** callback.

promise

The jQuery `promise` function can be used to obtain a promise against the completion of actions on a selection of DOM elements. Here's an example, adapted from the [.promise\(\) documentation](#):

```
$('#div').each(function(i){
    $(this).fadeIn().fadeOut(1000 * (i + 1));
});

$('#div').promise().done(function(){
    // All <div> animations are finished.
});
```

By default, the `promise` method will return a promise from a deferred that is resolved when all animations on an element collection have fired. You can also pass an argument to `promise` to name a different event queue that the promise should correspond to (the default name is “fx,” for “effects”). An even more advanced method is to pass an object to `promise`, in which case jQuery will attach the usual promise methods (`done`, `fail`, etc.) to the object and return it. These nondefault uses of `promise` are outside the scope of this book. To learn more, see the [.promise\(\) documentation](#) and hunt for examples online.



The `promise` function is only available client side: it operates on a collection of elements jQuery has selected from the DOM.

then

`then` is the only mechanism jQuery provides to take output values from one deferred, modify them, and plug them in as inputs to another deferred. You cannot do this with `done`, `fail`, or `progress`: any values returned by those callbacks are ignored. To properly understand what `then` does, you need to know that *it creates a new deferred* and returns its promise. Behind the scenes, it hooks up the output of the original promise to go through the modifying functions you provide and to then trigger the new deferred it has created (whose promise it will return).

The modification functions you give to `then` can themselves return new promises.

You can pass `then` up to three positional function arguments, to process a resolve value, an error value, or progress values from the original promise. Pad the call to `then` with `null` arguments, if necessary, to ensure that your function is in the correct position.

Here is a fanciful example that passes all three callback functions to `then`:

```

promise.then(
  function(result){
    // The deferred was resolved with result. Double it and pass it on.
    return 2 * result;
  },
  function(error){
    // The deferred was rejected with error. Log it and pass on a different
    // (null) error.
    console.log('Error received:', error);
    return null;
  },
  function(value){
    // The deferred made progress. Convert it to a percentage string.
    return Math.round(value * 100.0) + '%';
  }
);

```

Below is an example where only progress values from the original promise are modified and passed on. Any resolve or reject value will be passed through unchanged to the new promise, which logs progress and the eventual resolved value. Don't worry if you don't understand the calls to promise, notify, and resolve; you'll learn about them in “[Creating Deferreds](#)” on page 15.

```

var d = $.Deferred(),
    promise = d.promise();

var newPromise = promise.then(
  null,
  null,
  function(value){
    // The deferred made progress. Convert to a percentage string.
    return Math.round(value * 100.0) + '%';
  }
);

newPromise.progress(function(value){
  console.log('Progress:', value);
});

newPromise.done(function(value){
  console.log('Finished:', value);
});

d.notify(0.141592);
d.notify(0.618033);
d.resolve(27);

```

The preceding code produces the following output on the console:

```

Progress: 14%
Progress: 62%
Finished: 27

```



Although the call signature of the `then` method of jQuery promises resembles the `then` method of promises that follow the **Promises/A+ specification**, the two functions behave quite differently when it comes to processing errors and handling exceptions. In addition, the Promises/A+ `then` function only accepts two arguments. We'll discuss the differences in **"Promises/A+" on page 78**.

state

New deferreds start life in a pending state and remain that way until they are either resolved or rejected. Once resolved or rejected, their state can no longer change.

You can call **state** to discover the state of a deferred (or its promise). You'll get back a string, one of `'pending'`, `'resolved'`, or `'rejected'`. Unless you're debugging, you're unlikely to need this function.

when

jQuery has a **when function** to help you work with multiple promises. In this book we'll refer to it as `$.when`, which is how you'll likely use it in your code. Note that `$.when` is a totally separate top-level jQuery function; it's not part of a deferred or a promise.

If you have to call several functions that return promises and take an action only when all are completed, `$.when` is your friend.

Before we see how convenient `$.when` is, though, you can learn a lot by thinking about how it might work and by writing a version of it yourself, no matter how primitive or restricted. This should remind you strongly of the `makeBreakfast` challenge in **"Food for Thought" on page 1**.

Here's how you might naively write code to take an action after two promises are finished without using `$.when`:

```
function callAfter(firstAction, secondAction, finalize){
    // Call finalize after the deferreds returned by firstAction and
    // secondAction have both finished. Pass finalize the result from both
    // deferreds.

    // NOTE: Don't write code like this! Use $.when instead.

    var finishedCount = 0, result1, result2;

    firstAction().done(
        function(result){
            finishedCount++;
            result1 = result;
            if (finishedCount === 2){
                finalize(result1, result2);
            }
        }
    );
    secondAction().done(
        function(result){
            finishedCount++;
            result2 = result;
            if (finishedCount === 2){
                finalize(result1, result2);
            }
        }
    );
}
```

```

        }
    }
};

secondAction().done(
    function(result){
        finishedCount++;
        result2 = result;
        if (finishedCount === 2){
            finalize(result1, result2);
        }
    }
);
}

```

There are some serious problems with the above!²

1. There is no error checking. If `firstAction` or `secondAction` return a promise that is ultimately rejected, this is not handled.
2. What if we want to call three promise-returning functions, not two? More code duplication is needed and existing calls to `callAfter` will need to be found and changed.
3. What happens if `firstAction` or `secondAction` doesn't return a promise?
4. The return value of `finalize` is lost.

In spite of these problems, our code contains the core of what's needed. It keeps count of how many promises have been resolved and keeps the results for the eventual calling of `finalize`. Most importantly, *it gets its work done in a callback it adds to each promise it needs to monitor*. That's a pattern you'll encounter repeatedly in this book and in deferred-using code in the wild. It's part of what makes deferreds so cool.

The jQuery `$.when` function (with some help from `then`) takes care of all these issues, allowing you to simply write:

```
$.when(firstAction(), secondAction()).then(finalize);
```

The call to `$.when` returns a new promise, as does the call to `then`. If either `firstAction` or `secondAction` rejects the promise it returns, the error value will be propagated to `finalize`.

Can you see why the final result, from `finalize`, is no longer lost? When it becomes available, it will be used to resolve the promise returned by `then`. So if you need the result, you can simply write:

2. Later, in [“when2: An Improved jQuery.when” on page 63](#), we'll write a much more robust function that doesn't have these issues.

```
$.when(firstAction(), secondAction()).then(finish).done(function(){
    // arguments holds the result returned by the finish function.
});
```

\$.when also correctly handles the case in which the value returned by either (or both) of firstAction or secondAction is not a deferred.



There's a gotcha to watch out for when using \$.when. The following doesn't behave as you might hope:

```
var promises = [
    $.ajax('http://google.com'),
    $.ajax('http://yahoo.com'),
    $.ajax('http://www.nytimes.com')
];

// NOTE: DON'T DO THIS!
$.when(promises).then(function(results){
    console.log('All URLs fetched.');
```

There are two problems here. First, \$.when expects promises to be given as individual parameters, not as an array. Second, the results of the promises will be passed by \$.when as individual parameters to the callback function, not as a single array of results.

To pass an array of promises to \$.when, you'll therefore need to use apply:

```
var promises = [
    $.ajax('http://google.com'),
    $.ajax('http://yahoo.com'),
    $.ajax('http://www.nytimes.com')
];

$.when.apply($, promises).then(
    function(result1, result2, result3){
        console.log('All URLs fetched.');
```

Creating Deferreds

Once you're comfortable receiving promises from functions and working with them by attaching callbacks, you'll soon want to write your own function or API that creates a deferred and returns its promise.

Construction

To create a deferred, you just call \$.Deferred():

```
var deferred = $.Deferred();
```

As discussed in “[Terminology: Deferreds and Promises](#)” on page 3, instead of returning a deferred to the caller of your code, you’ll almost always want to return a promise. The promise is obtained by calling the `promise` method on the deferred:

```
var deferred = $.Deferred(),
    promise = deferred.promise();
```

For convenience, the `$.Deferred` constructor allows you to pass it a function that can be used to initialize the deferred once it is created. The function receives the deferred as its only argument. For example, here’s a handy `succeed` function³ that returns a promise that has already been fired with a specified result:

```
function succeed(value){
    return $.Deferred(function(d){
        d.resolve(value);
    }).promise();
}
```

Can you think of situations in which `succeed` might be useful? (Don’t worry if not—keep reading.)

You may find it interesting to read the official API documentation on [jQuery.Deferred\(\)](#).

resolve and resolveWith

You normally create a deferred and return its promise to your caller because you want to arrange for a slow task (such as a network or other I/O call) to be performed. When the task completes, you use the **resolve method** on the deferred to trigger the calling of any `done` or `always` callbacks attached to the promise.

The `resolve` call signature is dead simple: just call it with any arguments you like. These will be passed to all `done` and `always` callbacks added to the deferred’s promise.

It is perfectly acceptable, and quite common, to resolve a deferred before returning its promise. This tends to happen in situations where you write a function that normally causes a long-running task to be initiated but which sometimes already knows the answer, as we’ll see in “[Memoization](#)” on page 49. You can keep your function interface consistent (i.e., always returning a promise) by returning a promise that corresponds to a deferred you have already resolved. One beauty of deferreds is that your caller does not need to know whether the promise you return has already been resolved—they just add callbacks to it with no regard for the timing of resolution or rejection.

The variant, **resolveWith**, allows you to also pass a context object, which is then provided to the `always` and `done` callbacks as their `this` object.

3. This is named after the **Twisted** Python **function** of the same name.

reject and rejectWith

To indicate to a promise that an error has occurred (thus invoking its `always` and `fail` callbacks), use the **reject method** on the deferred. As with `resolve`, you may pass any arguments you like to the `reject` call. These will be passed to all `fail` and `always` callbacks added to the deferred's promise.

As with deferred resolution, it is perfectly acceptable to reject a deferred before returning its promise.

The variant, **rejectWith**, allows you to also pass a context object, which is then provided to the `always` and `fail` callbacks as their `this` object.

notify and notifyWith

After you've created a deferred and returned its promise, you'll at some later point want to resolve or reject the deferred so your caller (who has the promise) can proceed. In some circumstances, though, you might obtain information about the progress of the underlying operation that you launched, and may want to pass that information along to the promise holder. The classic UI example is an animated progress bar that shows the user what percentage complete a task is.

The **notify call signature** is also dead simple: just call `notify` with any arguments you like. These will be passed to all progress callbacks added to the deferred's promise.

And yes, you guessed it, the **notifyWith variant** allows you to also pass a context object, which is then provided to the progress callbacks as their `this` object.

Note that once you have resolved or rejected a deferred, further `notify` calls on it will have no effect (i.e., they will not result in invocation of any progress callbacks on the promise).

Putting It All Together

Now that you know all about creating deferreds and consuming promises, it should be clear that there are three pairs of methods on a deferred that you can invoke that cause the running of a corresponding set of callbacks on the promise for the deferred. **Table 2-1** shows the naming correspondence.

If you call a deferred method named in the left column with some set of arguments, all the callbacks added via the promise method named in the right column will be invoked, in the order they were added, each with the same set of arguments.

Table 2-1. Correspondence between deferred and promise method names

Deferred method	Promise method
<code>resolve</code> or <code>resolveWith</code>	<code>done</code>
<code>reject</code> or <code>rejectWith</code>	<code>fail</code>
<code>notify</code> or <code>notifyWith</code>	<code>progress</code>

Deferred Dynamics

Here is a summary of aspects of deferred behavior (some of which were described at length earlier) that you need to understand, along with some principles of deferred usage:

- If your code creates a deferred but doesn't eventually cause it to be resolved or rejected, you're probably doing it wrong. Almost without exception, code that creates a deferred should be responsible for getting it fired.
- A function that creates a deferred can resolve or reject it before its promise is returned.
- If you write a function that creates a deferred that you want to return, return the result of calling `promise()` on the deferred.
- Any `always`, `done`, or `fail` callbacks added to a promise whose deferred has already been resolved or rejected will be called immediately, as appropriate.
- The `resolve`, `reject`, and `notify` methods of a deferred can be called with any number of arguments, all of which will be passed to the corresponding methods on the promise.
- When a deferred is resolved (or rejected), the `done` (or `fail`) callbacks of its promise are invoked in the order they were added.
- `always` callback functions are appended to both the `done` and `fail` lists of functions to be called when the deferred fires. So they will be called, in turn, as the functions in the list of `done` or `fail` callbacks is invoked.
- Fire at will! It's not an error to resolve or reject a deferred multiple times. As you'd expect, only the first resolve or reject triggers callbacks attached to the promise. You don't need to keep track of whether a deferred you create has been fired. The deferred handles that for you.
- If you call `notify` on a deferred that has already been resolved or rejected, nothing happens (i.e., any `progress` callbacks attached to the deferred's promise will not be called).
- It is perfectly safe to return the same promise instance to multiple independent callers of your code. Any `done`, `fail`, or `progress` callbacks added to it cannot

interfere with one another. In other words, there is no chaining of returned values from `done`, `fail`, or `progress` callbacks.

- Only `then` can be used to pass on a modified return value from a deferred. *It does this by creating a new deferred* and returning its promise.
- If, as a promise *consumer*, you find yourself wondering about whether a promise has fired or not (i.e., your code relies on **state**), you probably don't fully understand deferreds. Part of the point is that you don't have to worry.
- Use `reject` for low-level, fundamental, exceptional problems (i.e., where you'd raise an exception in nondeferred code). For example, if you make a call to a remote HTTP service, use `reject` for the cases where there's a network error. Use `resolve` to send back a status code (even if it's an error code) and a result.

Deprecated Promise Methods

We'll mention the deprecated promise methods in case you ever have to maintain older jQuery-based code. If possible, replace them with the suggested alternatives.

isRejected and isResolved

isRejected and **isResolved** returned Boolean values to indicate if a deferred or promise is in a rejected or resolved state. They were deprecated in jQuery 1.7 and removed in 1.8. Use the new **state method** instead.

pipe

pipe is the now-deprecated (in jQuery 1.8) original name of the **then method**. The two are identical: a quick look at *deferred.js* reveals the line `promise.pipe = promise.then;`

Changes in the jQuery Deferred API

Be careful when reading the jQuery **Deferred object documentation**. Deferreds were introduced to jQuery in version 1.5 and in several subsequent versions there were changes to parts of the deferred API. For example, in the documentation for **then**, you'll see its supported call signatures changed in versions 1.7 and 1.8 and that the different behaviors are all still documented.

When reading code that uses jQuery deferreds, it is therefore sometimes important to know what version of jQuery the code will be run against. That advice applies to this book too (jQuery versions 1.10.2 and 2.0.3 are now current). And, needless to say (hopefully!), when you're writing deferred code you need to know what version of jQuery will be loaded.

Likewise, if you're modernizing an existing project that uses deferreds, be careful in moving to a new version of jQuery—especially if, through some inconceivable chain of events, the code you're working on doesn't have a test suite!

Deferred Recipes

In this chapter you will find many examples of using jQuery deferreds. These are all based on actual use cases. The intention is to show you a broad collection of real-world uses of deferreds in order to make this book maximally useful.

Although we've arranged the examples to be read in order of increasing difficulty, most of them can be read in isolation. We encourage you to “dip in” wherever it's useful.

A Replacement for the `setTimeout` Function

Sometimes, when using a site such as Twitter or Gmail, a warning message appears when an asynchronous task is taking too long. For example, Twitter will replace the “spinner” at the bottom of their infinite scroll with an “oops” message along with a suggestion that you click a link to trigger the request again or refresh the page. Gmail is cleverer: when there is a connectivity problem, it displays a warning message and then counts down until the point at which it retries the network call that failed.

There is a built-in function in JavaScript called `setTimeout`. It works like this:

```
setTimeout(function(){  
    console.log('Display after half a second');  
}, 500);  
console.log('Display immediately (unblocked)');
```

The result of this is:

```
Display immediately (unblocked)  
Display after half a second
```

Implementing the functionality described above using just `setTimeout` might be hard to do in a clear and comprehensible way, especially if there were the potential for nested calls to `setTimeout`.

This kind of thing is easy to achieve with deferreds. In fact, the Gmail example almost reads as if it is describing a deferred: “when there is a connectivity problem, display a warning message and then retry after a certain number of seconds.”

Here’s a simple replacement for `setTimeout` that uses deferreds. This is adapted from *Fun With jQuery Deferreds* by Michael Bleigh.

```
function wait(timeout){
    var deferred = $.Deferred();
    setTimeout(deferred.resolve, timeout);
    return deferred.promise();
}
```

And here’s how you use it:

```
wait(500).done(function(){
    console.log('Timeout fired!');
});
```

This produces the following, as you would expect:

```
Timeout fired!
```

Given a promise-based timeout, it is not too hard to combine it with other promises to create solutions similar to those implemented by Twitter and Gmail. We’ll show you how you might do this in “[Using when2 to Time Out a Single Promise](#)” on page 67, once we have another handy utility in our deferred tool belt. Furthermore, the promise returned by the `wait` function is a representation of the timeout that can be reused and passed around your code in ways that would be impossible if you’d stuck to the plain old `setTimeout` function.

Challenges

1. As you may remember from reading “[Construction](#)” on page 15, you can pass a function to `$.Deferred`. The function will be called with the new deferred instance. This is just a convenience; you can create and configure a deferred in one go. Change `wait` to work in this way.
2. Suppose there were functions called `one` and `two` that you wanted to call when the timeout expired. You could put calls to them into the function you pass to `then`. Subtly different though would be to instead somehow pass the timeout to those other functions. How would you do that?
3. `setTimeout` returns a value that can be passed to `clearTimeout` to prevent the timeout function from being run. Our `wait` function lacks this capability. Change it so that it returns a promise that also has a `cancel` method on it that will cause the promise returned by `wait` to be rejected.

Messaging in Chrome Extensions

A common need when working on a project that uses deferreds is to use an existing callback-based API in a way that's compatible with deferreds. This is usually a simple task.

Here we'll illustrate the process using the messaging API found in Chrome extensions. Don't worry if you don't use Chrome or don't build extensions with it; this example doesn't require any prior Chrome experience.

A Chrome extension can have an invisible “background” page that communicates with “content scripts” running in the tabs the user has open. The API call used by the background page to send a message to a tab is `chrome.tabs.sendMessage`, with the following signature:

```
chrome.tabs.sendMessage(tabId, message, responseCallback);
```

Here the message can be anything at all. The `responseCallback`, if provided, will be called with a response if the content script running in the tab wants to send back a reply, or will be called with no argument in the case of a Chrome error (e.g., trying to send a message to a nonexistent tab). So a typical nondeferred usage might be:

```
chrome.tabs.sendMessage(17, { action: 'addSidebar' }, function(result){
  if (result === undefined){
    console.error('Chrome error:', chrome.runtime.lasterror.message);
  }
  else if (result.success){
    console.log('Sidebar added to DOM.');
```

To convert this to code that uses deferreds, we'll first need a replacement for `chrome.tabs.sendMessage` that returns a promise:

```
function sendMessageDeferred(tabId, message){
  var deferred = $.Deferred(); // ❶

  chrome.tabs.sendMessage(tabId, message, function(result){
    if (result === undefined){
      deferred.reject(chrome.runtime.lasterror.message); // ❷
    }
    else {
      deferred.resolve(result); // ❸
    }
  });

  return deferred.promise(); // ❹
}
```

You'll use this pattern over and over again as you work with deferreds.

- ❶ Make a deferred, whose promise will be returned to the caller.
- ❷ Arrange to reject the deferred if anything goes wrong in making the API call(s) you need to make.
- ❸ Arrange to resolve the deferred with a result value if all goes well.
- ❹ Return the deferred's promise to our caller.

With the `sendMessageDeferred` function in place, we can now use it in our code:

```
sendMessageDeferred(17, { action: 'addSidebar' })
  .done(function(result){
    if (result.success){
      console.log('Sidebar added to DOM.');
```

We could use `$.when` to send messages to several tabs, only proceeding once all tabs had replied:

```
$.when(
  sendMessageDeferred(17, { action: 'addSidebar' }),
  sendMessageDeferred(18, { action: 'hideSidebar' }),
  sendMessageDeferred(19, { action: 'hideSidebar' })
)
  .done(function(){
    console.log('All tab sidebars adjusted.');
```

As we saw in “[when](#)” on page 13, it's quite awkward to log a status line (as in the `done` callback above) after sending a message to multiple tabs without using deferreds. Further, in this simple example, the set of tabs to send to is static. Writing this code without deferreds is *much* more difficult if the set of tabs to send to is stored in a list of arbitrary size.

Challenges

1. What will happen if the content script in the tab receiving the message does not call the `sendResponse` function?

2. If the tabs wanted to return a result (for example, the elapsed time taken to hide or show the sidebar), how would we access the three results?

Accessing Chrome Local Storage

As a second example of converting a callback API to a deferred API, let's write a deferred-based API for access to the Chrome browser's local storage.

The Chrome local storage API differs from the W3C Web Storage `localStorage` API in some important ways. Of most interest to us is that the Chrome local storage is asynchronous. The methods used to get, set, etc., keys each take a callback function that is called once the storage operation has completed. You can check the [chrome.storage documentation](#) for the complete details of the API.

Making a deferred API from the Chrome API is very easy. There are five functions: `get`, `getBytesInUse`, `set`, `clear`, and `remove`. They all have similar signatures. For example, `set` looks like this:

```
chrome.storage.local.set(items, callback)
```

To make a deferred method for `set`, we don't really need to know what `items` represents. All we need to know is that `callback` will be called by Chrome when it has set the key values it is given. Our version of `set` will look as follows:

```
function set(items){
  var deferred = $.Deferred(); // ❶

  chrome.storage.local.set(items, function(){
    if (chrome.runtime.lasterror){
      deferred.reject(chrome.runtime.lasterror.message); // ❷
    }
    else {
      deferred.resolve(); // ❸
    }
  });

  return deferred.promise(); // ❹
}
```

This should look familiar by now. It follows the classic pattern:

- ❶ Create a deferred.
- ❷ Arrange to reject the deferred in case of error.
- ❸ Arrange to resolve the deferred with a result if no error occurs.
- ❹ Return a promise created from the deferred.

You'd use the above `set` function as follows:

```
set({
  lastURL: 'http://bit.ly/jquery-deferreds',
  timestamp: new Date().toUTCString()
}).done(function(){
  // Keys for the user's last URL and timestamp saved to local storage.
});
```

Challenges

1. Write deferred versions for the other `chrome.storage.local` functions (`get`, `getBytesInUse`, `remove`, and `clear`).
2. The `node.js Redis package` provides a callback-style interface to the `Redis`. Choose some Redis API calls and make deferred versions of them. Your functions should call the existing callback-based API to get the Redis work done; you're just providing a deferred interface.

Running Promise-Returning Functions One by One

There will be occasions when you need to call several promise-returning functions one by one, with each function starting only after the preceding one has completed. Sometimes implicit dependencies between the functions require this, but more commonly you'll just need to use the result of an early function as an input to a later one.

If you read [Chapter 2](#), you'll already know how to do this by using the `then` method of a returned promise. For example, if you want to change the opacity on one DOM element and only then change the opacity on a second element, you could write:

```
var promise = $('#label-1').animate({ opacity: 0.25 }, 100).promise()
  .then(function(){
    return $('#label-2').animate({ opacity: 0.75 }, 200).promise();
  });
```

While this doesn't look too bad for running two functions one after another, it quickly gets uglier with more. A more serious drawback is that because it is just static code, it can't be used if we have a list of promise-returning functions we need to run one after the other.

It would look much nicer—and be easier to add to—if we could write something like this:

```
var promise = synchronously([
  [ $('#label-1').animate, { opacity: 0.25 }, 100 ],
  [ $('#label-2').animate, { opacity: 0.75 }, 200 ]
]);
```

Fortunately, it's easy to write synchronously, a function that takes a list of functions to run, along with their arguments. It returns a promise that fires when all functions have been run to completion one after another.

Here's how it might look:

```
function synchronously(tasks){
  var i, task, func,
      promise = $.Deferred().resolve().promise(), // ❶
      makeRunner = function(func, args){ // ❷
        return function(){
          return func.apply(null, args).promise();
        };
      };
  for (i = 0; i < tasks.length; i++){
    task = tasks[i];
    func = task.shift();
    promise = promise.then(makeRunner(func, task)); // ❸
  }
  return promise;
}
```

This works as follows:

- ❶ Create a promise variable that we'll eventually return. The deferred from which the promise is made is resolved immediately with an undefined value. When we first call then on that promise, the function we pass to then will be run immediately. Note that we don't care about the value the promise fires with; we're just trying to get the task functions run sequentially.

- ❷ makeRunner functions as a closure that returns a function to invoke the current task with its arguments. Note that we assume that calling func returns either a promise or something that can be used to obtain a promise.

There is an unfortunate subtlety here. All promises have a `promise` method (undocumented!) that, when called with no arguments, returns the promise. The method is actually *identical* to the (documented) `promise` method of its deferred. So if func returns a promise, calling `promise()` on it simply returns the promise. More usefully, if func happens to be a call to `animate`, the call to `promise()` is essential—it ensures that the returned value, passed to then, is a promise. See the challenges below for more on this.

- ❸ This slightly dubious line reuses (updates) the promise variable to hold the promise returned by calling then on the current promise. In effect, we're adding a new function to be called when the current promise is done and then forgetting about that promise. We only have to worry about the most recent promise, which we return to our caller. Because the original promise was fired on creation, we don't have to hang onto it for any reason.

Challenges

1. Improve the `synchronously` function so that each function is run with a passed `this` object for its context.
2. The anonymous function passed to `promise.then` in `synchronously` ignores any arguments it is passed. Why is that? What will its arguments be?
3. How could you improve `synchronously` to allow for the case in which some of the functions to be called did not return promises or objects with a `promise` method?
4. The following code fragment is identical to the first one above, but it uses `done` instead of `then`:

```
var promise = $('#label-1').animate({ opacity: 0.25 }, 100).promise()
  .done(
    function(){
      return $('#label-2').animate({ opacity: 0.75 }, 200).promise();
    }
  );
```

What difference will this make to code that adds callbacks to the promise? The answer is important. If you don't know it, you don't properly understand jQuery deferreds yet.

5. This fragment uses `when` instead of `then`:

```
var promise = $.when(
  $('#label-1').animate({ opacity: 0.25 }, 100),
  $('#label-2').animate({ opacity: 0.75 }, 200)
);
```

How will this behave differently from our earlier two fragments that used `then` and `done`? Why is it not necessary to call `.promise()` on the result of the call to `animate`?

A Promise Pool with an emptyPromise Method

When running a node service, you'll occasionally need to shut it down. The service might have many requests in progress, so ideally, you should shut things down gracefully. Some of the outstanding requests may have users on the other ends of them. When the shutdown signal arrives, you may need to make several calls to cleanly disconnect from other services or to start others.

In these cases, a useful and simple trick is to maintain a pool of promises that correspond to outstanding tasks, and to provide a method that returns a promise that fires when the pool is empty (i.e., when all the promises in the pool have completed).

Creating a Promise Pool

Here's `createPromisePool`, a function that maintains a promise pool as described above. It returns an object with `add` and `emptyPromise` methods. The former adds a promise to the pool and the latter returns a promise that resolves when the pool empties:

```
var $ = require('jquery-deferred'); // ❶

function createPromisePool(){

  var inProgress = [], // ❷
      waitingForEmpty = []; // ❸

  return {
    add: function(promise){
      inProgress.push(promise); // ❹
      promise.always(function(){
        var i;
        for (i = 0; i < inProgress.length; i++){ // ❺
          if (inProgress[i] === promise){
            inProgress.splice(i, 1);
            break;
          }
        }

        if (inProgress.length === 0){ // ❻
          for (i = 0; i < waitingForEmpty.length; i++){
            waitingForEmpty[i].resolve();
          }
          waitingForEmpty = [];
        }
      });
    },

    emptyPromise: function(){ // ❼
      var deferred = $.Deferred();
      waitingForEmpty.push(deferred);
      return deferred.promise();
    }
  };
}
```

- ❶ `require` is a core node.js function for loading JavaScript from another file.
- ❷ `inProgress` holds a list of outstanding promises.
- ❸ `waitingForEmpty` holds a list of deferreds whose promises have been given to pool observers that want to know when the pool is empty.
- ❹ Add the given promise to the list of in-progress tasks.
- ❺ When the promise fires, remove it from the in-progress promises.

- ⑥ If we now have no promises in progress, resolve all waiting deferreds.
- ⑦ The `emptyPromise` function creates a new deferred, adds it to the list of waiters, and returns its promise.

Using the Promise Pool

Suppose you're using the **Express** node HTTP server and you have a function called `handleRequest` that takes an incoming request and returns a promise that fires once the request has been dealt with. Server code that uses a promise pool might have fragments like this:

```
var express = require('express'),
    app = express(),
    pool = createPromisePool(),
    shuttingDown = false;

app.get('/', function(req, res){
  if (shuttingDown){
    res.redirect('http://example.com'); // ①
  }
  else {
    pool.add(handleRequest(req, res)); // ②
  }
});

app.get('/shutdown', function(req, res){
  if (shuttingDown === false){
    shuttingDown = true;
    pool.add(flushAvatarCache()); // ③
    pool.add(sendShutdownEmail());
    pool.add(flushLogs());
    pool.emptyPromise().done( // ④
      function(){
        res.send(200); // ⑤
        process.exit(0);
      }
    );
  }
  else {
    res.send(200); // ⑥
  }
});

app.listen(9999);
```

- ① If the server is shutting down when a request arrives, redirect the request to some other URL (alternatively, we could fail the request). The important thing is to stop processing requests—to not add any more request promises to the pool.

- ❷ Otherwise, call `handleRequest` and add the promise it returns to the pool.
- ❸ On shutdown, arrange for final tasks to be done, adding the promise for each one to the pool of outstanding work.
- ❹ When all outstanding work is finished, exit the node process.
- ❺ The server status (200) is not written back to the client until the service has really been halted. We could have returned a 200 status immediately, but by waiting until everything is done, the client knows their call succeeded.
- ❻ If a shutdown request arrives while we are already shutting down, we simply return a 200 immediately (see the challenges below).

Challenges

1. Change the `emptyPromise` function so it consists of a single `return` statement. Hint: pass an initialization function to `$.Deferred`.
2. What will happen if the in-progress pool is empty when `emptyPromise` is called?
3. Change `emptyPromise` to accept a Boolean argument, `checkImmediately`. If `checkImmediately` is true and the in-progress pool is empty when `emptyPromise` is called, it should return an already resolved promise.
4. The processing of a shutdown request is too simplistic because it may start functions like `flushAvatarCache` before all existing HTTP requests are finished. Change the shutdown code so that it first waits for the pool of HTTP GET requests to empty, then adds shutdown promises to the pool, and then waits again for the pool to empty before exiting.
5. Change `createPromisePool` to allow the caller to pass a function. The function should be evaluated each time an outstanding promise fires and if it returns `true`, resolve all the waiting deferreds. You might like to pass the function the number of items in the `inProgress` list, or even the list itself (if you're the trusting type).
6. In `createPromisePool` the `inProgress` list is linearly scanned and then spliced (in the `always` callback) each time a request finishes. This will be very inefficient if you are processing many simultaneous requests. How could you change `inProgress` to be a JavaScript object so you could then use `delete` to quickly remove the completed promise? Don't forget to also change the `inProgress.length === 0` test.
7. If a shutdown request arrives while we are already shutting down, the code above returns a 200 immediately. This might be problematic: such a 200 response actually only indicates that a shutdown is in progress, not that it has completed. A better approach would be to only return a 200 when the currently running shutdown process has completed. Change the shutdown code so that additional shutdown requests receive a 200 response only when the in-progress shutdown sequence

completes. Your code should work no matter how many shutdown requests are received during the shutdown process.

Displaying Google Maps

The **Google Maps API** makes it extremely easy to add map images to web applications. Just add a `src` attribute containing a latitude and longitude of a location to an HTML `` tag and Google takes care of the rest.

The direct approach is extremely convenient and easy, but is likely too simplistic in practice. One issue is that network requests are slow, particularly on mobile devices. Also, the Google API is no longer free or may be limited by a quota, so reducing API calls by caching can save money and/or keep an application working correctly for longer. Caching can have strong benefits for location-aware applications since many of us are creatures of habit, repeatedly visiting certain places like home, the office, the gym, and favorite restaurants.

In this section, we'll build a function that returns a function that manages a cache of fetched maps. As you'd expect, the returned function does its work without hitting the Google API unless necessary.

We'll end up being able to write code like this:

```
var fetchMap = createMapCache(),
    latitude = 52.203558,
    longitude = 0.119436;

fetchMap(latitude, longitude).done(function(map){
  // Use the map.
});
```

The first time you call `fetchMap` with a particular (latitude, longitude) pair, the map is fetched from Google. Thereafter, calls for the same location will quickly return a locally cached map.

We'll build a three-level solution, storing a **data URI** locally. If you're unfamiliar with them, data URIs are a nice way to easily store image data (as **base 64** strings) in memory in a format that can be used in HTML `` tags.

Our approach will store the coordinates and data URI of the last-requested location in memory as JavaScript variables. When called, our lookup function will first check the last coordinates and return the stored data URI if the location matches. If the in-memory location does not match, we'll check Chrome local storage. Only if that also fails will we go to the Google API. On any cache miss (either in-memory or local storage, or both) we'll make sure to save the newly retrieved data URI as appropriate.

The use of Chrome local storage for the middle layer of our approach is an unimportant detail. Please don't let it put you off if you're not a Chrome user. The example is worth considering in any case, and you should be able to swap in your own middle level if necessary.

First of all, here are a couple of utility functions to get and set values from Chrome local storage. We covered this topic in [“Accessing Chrome Local Storage” on page 25](#), so these functions should look familiar.

```
function localStorageGet(keys){
    var deferred = $.Deferred();
    chrome.storage.local.get(keys, function(result){
        if (chrome.runtime.lasterror){
            deferred.reject(chrome.runtime.lasterror.message);
        }
        else {
            deferred.resolve(result);
        }
    });
    return deferred.promise();
}

function localStorageSet(items){
    var deferred = $.Deferred();
    chrome.storage.local.set(items, function(){
        if (chrome.runtime.lasterror){
            deferred.reject(chrome.runtime.lasterror.message);
        }
        else {
            deferred.resolve();
        }
    });
    return deferred.promise();
}
```

We'll also need a promise-returning function that does a Google Maps API lookup of a map, given a latitude and longitude:

```
function googleLookup(latitude, longitude){
    var deferred = $.Deferred(),
        url = ('http://maps.googleapis.com/maps/api/staticmap?center=' + // ❶
            latitude + ',' + longitude +
            '&zoom=10&size=400x200&sensor=false'),
        TIMEOUT = 1500,
        xhr = new XMLHttpRequest();

    xhr.open('GET', url, true);
    xhr.send();
    xhr.responseType = 'arraybuffer'; // ❷

    var requestTimer = setTimeout(function(){ // ❸
        xhr.abort();
    }, TIMEOUT);
    deferred.resolve(xhr.response);
    return deferred.promise();
}
```

```

        deferred.reject();
    }, TIMEOUT);

    xhr.onload = function(){
        clearTimeout(requestTimer);
        if (xhr.status === 200){
            deferred.resolve(
                'data:image/png;base64,' +
                base64ArrayBuffer(xhr.response) // ❷
            );
        }
        else {
            deferred.reject();
        }
    };

    xhr.onerror = function(){
        clearTimeout(requestTimer);
        deferred.reject();
    };

    return deferred.promise();
}

```

- ❶ The URL here is the endpoint for the [Google Maps API](#).
- ❷ We create our own XMLHttpRequest object so that we can set its responseType to arraybuffer. Normally we'd use a \$.ajax call for convenience, but the dataType option for \$.ajax does not currently allow for receiving an ArrayBuffer.
- ❸ We set up a timeout to reject the deferred if the Google API call doesn't complete within TIMEOUT milliseconds.
- ❹ When the data becomes available from Google, we turn it into a data URI. The base64ArrayBuffer function that converts an ArrayBuffer to base 64 is given in [Appendix C](#).

Finally, here's the main function, createMapCache. It returns a map-fetching function that maintains a cache:

```

function createMapCache(){

    var lastKey, lastDataURI; // ❶

    return function(latitude, longitude){ // ❷
        var deferred = $.Deferred();

        var getMap = function(latitude, longitude){ // ❸
            var key = 'map-' + latitude + '-' + longitude; // ❹

            if (key === lastKey){ // ❺
                deferred.resolve(lastDataURI);
            }
        };

        $.ajax({
            url: 'https://maps.googleapis.com/maps/api/staticmap?center=' +
                latitude + ',' + longitude + '&size=256x256&key=' +
                'AIzaSyB5I3FpUqBnuvL3iWnFnuqumYn6HddHn60',
            dataType: 'arraybuffer',
            success: getMap,
            error: getMap,
            timeout: 10000
        })
        .done(function(){
            lastDataURI = deferred.resolve().value();
        })
        .fail(function(){
            deferred.reject();
        });

        return deferred.promise();
    };
}

```

```

    }
    else {
        localStorageGet(key).then( // 6
            function(obj){
                if (obj.hasOwnProperty(key)){ // 7
                    lastDataURI = obj[key];
                    lastKey = key;
                    deferred.resolve(lastDataURI);
                }
                else {
                    // key not in local storage, use Google.
                    googleLookup(latitude, longitude).then(
                        function(dataURI){
                            var store = {};
                            store[key] = dataURI;
                            localStorageSet(store); // 8
                            lastDataURI = dataURI;
                            lastKey = key;
                            deferred.resolve(dataURI);
                        },
                        function(error){
                            deferred.reject(error);
                        }
                    );
                }
            },
            function(error){
                deferred.reject(error);
            }
        );
    }
};

if (arguments.length === 2){ // 9
    getMap(latitude, longitude);
}
else {
    navigator.geolocation.getCurrentPosition( // 10
        function(position){
            getMap(position.coords.latitude, position.coords.longitude);
        },
        function(error){
            deferred.reject(error);
        }
    );
}

return deferred.promise();
};
}

```

- ❶ `lastKey` keeps track of the last key we used (in local storage) so we can reply quickly to a repeated request. Similarly, `lastDataURI` is the value associated with the last key.
- ❷ Return a function that takes a latitude and a longitude.
- ❸ `getMap` is the function that does all the work.
- ❹ First, we choose a key for local storage. The `map-` prefix aims to reduce the risk of key collision with other apps.
- ❺ The first check is a simple synchronous test against the location we were called with last time.
- ❻ If the in-memory lookup doesn't work, arrange to look in the local storage.
- ❼ If we find the key in local storage, remember it and return its value via resolving the deferred.
- ❽ Otherwise, the key was not in local storage so we call the Google Maps API. If that is successful, we store the new value in local storage and in the `lastKey` and `lastDataURI` variables.
- ❾ The function we're returning can accept latitude and longitude arguments, so if we're provided with these, call `getMap` directly.
- ❿ Otherwise, get the user's latitude and longitude from the browser. We assume here that you have a modern browser, with support for `navigator.geolocation`.

Challenges

1. Storing just one last data URI in memory is pretty stingy. Add an `LRU data structure` to store a larger number of recent results.
2. `getMap` is the only function in this book that calls a promise-returning function but ignores the returned promise. Can you see where? Why is the promise ignored?
3. Error handling in the above could be greatly improved. Errors can arise when asking the user to permit location information for the application, in talking to Chrome's local storage, and (for several reasons) in the Google Maps API. Add a well-defined error object that the deferred is rejected with that will always allow the caller to detect what went wrong.
4. One shortcoming of the above code is that it may make unnecessary simultaneous calls to the Google Maps API. Suppose a first call triggers a query to the Google API and that while that network request is still in progress, a second call to our function arrives for the same location. The second call will trigger an identical request to the Google API. This is not too hard to prevent. Add a JavaScript object that will store deferreds from requests in progress. When a new call arrives, check to see if the coordinates correspond to an outstanding request and, if so, return its promise.

You'll need to add cleanup code via a callback on each deferred you create to take the deferred out of the set of in-progress deferreds.

This is an excellent example of a common pattern: put deferreds into a data structure as you create (or receive) them and add callbacks to them to remove them from the data structure after they fire. We saw this in [“A Promise Pool with an emptyPromise Method” on page 28](#) and will encounter it again in [“Short-Term Memoization of In-Progress Function Calls” on page 52](#).

Communicating with a Web Worker

JavaScript is single-threaded. Calling long-running functions, whether client side or server side, can have a severe usability impact. Client side, the browser UI freezes while the call is in progress. Server side, processing of other requests stops while the long-running function executes. New connections may be missed.

Client side, the traditional approach to alleviating this situation has been to perform computations in parts, giving the browser a chance to attend to other tasks. This is an unnatural approach to programming, often made more awkward by the use of `set Timeout` to schedule the next phase of a computation.

Web workers are designed to fix this problem. A web worker is simply a separate JavaScript thread, spawned by the main thread. Communication with the web worker is done using message passing. At any point, the main thread can send a message to a worker, and vice versa.

In this example, we'll make it possible to send a message to a web worker and receive a promise that will fire with the reply. This is very useful because the promise provides a formal mechanism for receiving a reply to a message. When your code makes a call to a web worker, it can immediately arrange for the processing of the response instead of fielding a reply elsewhere in your code (which will likely involve having to figure out what it's a reply to, as well as reestablishing the context in which you originally sent the message).

The Web Worker Code

Below is the code that will be run in the web worker thread:

```
var METHODS = { // ❶
  add: function(callback, payload){
    callback({ result: payload[0] + payload[1] });
  },

  ping: function(callback){
    log('Ping command received, replying with pong.');
```

```
    callback({ result: 'pong' });
  }
};
```

```

    }
  };

  addEventListener('message', function(event){ // ❷
    var job = event.data;
    if (METHODS.hasOwnProperty(job.method)){
      METHODS[job.method](
        function(result){
          result.requestId = job.requestId;
          postMessage(result); // ❸
        },
        job.payload);
    }
    else { // ❹
      log("Attempt to call unknown method '" + job.method + "'.");
    }
  });

  function log(message){ // ❺
    postMessage({
      message: message,
      type: 'log'
    });
  }
}

```

- ❶ The METHODS object has a key for each of the methods that the main thread can call in the web worker. There are two functions: one to add two arguments passed in the payload and one to respond to a ping command with the string pong.
- ❷ The addEventListener section is the key to the coordination of incoming messages and outgoing results. It examines the incoming message for the method to call and calls the corresponding method (in the METHODS object) with a callback and the message payload. It takes the result of the call and sends it back to the main thread along with the request ID from the incoming message.
- ❸ The postMessage function is part of the web worker specification. It is used to send a message back to the code that created the web worker.
- ❹ It is possible that you might want your web worker to process messages that don't need replies and have nothing to do with deferreds, request IDs, etc. If so, those can easily be handled here. For now, we're just logging the reception of an unrecognized message.
- ❺ The log function illustrates that the web worker can still send an unsolicited message to the main thread. This will be used if an attempt is made to call an unknown method. The ping function also logs its usage.

Let's suppose we've stored this code in a file called *examples/src/web-worker.js*. We'll need the filename below.

Creating a Web Worker

The request ID in each message is obviously what ties together requests and their replies.

You can see how the ID is used in `createWebWorker`, below, which creates a web worker and arranges to send it messages. `createWebWorker` returns an object with `add` and `ping` methods on it, both of which return promises.

```
function createWebWorker(sourceFile){ // ❶

    var requests = {}, // Key is request id, value is a deferred.
        requestId = 0, // Incremented on each message sent.
        worker = new Worker(sourceFile),

    sendJob = function(method, payload){ // ❷
        var deferred = $.Deferred(),
            id = requestId++;
        requests[id] = deferred;
        worker.postMessage({
            method: method,
            payload: payload,
            requestId: id
        });
        return deferred.promise();
    },

    handleResponse = function(response){ // ❸
        var deferred, id;
        if (response.hasOwnProperty('requestId')){
            id = response.requestId;
            if (requests.hasOwnProperty(id)){
                deferred = requests[id];
                delete requests[id];
                if (response.hasOwnProperty('result')){
                    deferred.resolve(response.result);
                }
                else {
                    deferred.reject(response.error);
                }
            }
        }
        else {
            // An unsolicited message from the worker. ❹
            if (response.type === 'log'){
                console.log('Worker says:', response.message);
            }
            else {
                console.log('Unknown message from worker:', response);
            }
        }
    };
};
```

```

worker.addEventListener('message', function(event){ // ⑤
    handleResponse(event.data);
});

return { // ⑥
    add: function(a, b){
        return sendJob('add', [a, b]);
    },

    ping: function(){
        return sendJob('ping');
    }
};
}

```

- ❶ The name of the source file containing the web worker code must be passed to `createWebWorker`, which passes it to the `Worker` constructor.
- ❷ The `sendJob` function is used to send a message to the web worker. Each message specifies what method should be run by the web worker, its payload (which may be undefined), and a request ID. A deferred is created and stored in the `requests` object using the request ID. Storing the deferreds associated with the in-progress tasks requested of the web worker will allow us to pass the eventual result to our caller.
- ❸ The `handleResponse` function is used to process all incoming messages from the web worker. These are first checked to see if they contain a request ID. If so, the deferred corresponding to the request is taken from the `requests` object and it is either resolved or rejected depending on the presence of a result in the message.
- ❹ Incoming messages that do not have a request ID can also be handled, as illustrated by the `log` message. The `log` messages are a handy way of getting information out of a web worker (which does not have its own console object or other mechanism for providing diagnostics).
- ❺ Add an event listener that receives messages from the worker. All messages are passed to `handleResponse`.
- ❻ The `createWebWorker` function returns an object with `add` and `ping` attributes that can be used to call the corresponding function in the web worker. Both these methods return a promise, as you would (hopefully!) expect.

Using It

Finally, here's an example of how you could use the above:

```

var worker = createWebWorker('examples/src/web-worker.js'); // ❶

$.when(worker.add(3, 4), worker.ping()) // ❷

```



```

.done(
  function(total, pingReply){
    console.log('Total = ' + total + '. Ping reply = ' + pingReply + '.');
  }
);

```

- ❶ First, we create a web worker via a call to `createWebWorker`. The argument is the name of the file containing the web worker code.
- ❷ The jQuery `$.when` function is used to coordinate the result of the two promises obtained by calling `worker.add` and `worker.ping`.

The above produces the following output on the console log, as you would expect:

```

Worker says: Ping command received, replying with pong.
Total = 7. Ping reply = pong.

```

Summary

The above approach takes nothing away from the informal ad hoc messaging used between the main thread and its web workers. The main thread and the web worker are both still free to use `postMessage` at any time. Our code just adds some explicit methods that can be called in the web worker and arranges, in the main thread, for the corresponding functions to return promises. Any incoming message with a request ID is assumed to be a response to such a method call. Messages from the web worker without request IDs are handled separately (as with the `log` messages).

Challenges

1. Change `createWebWorker` to log a message if it receives a response containing a request ID that is unknown.
2. Let's add some timing information to `createWebWorker`. Change it to store a timestamp for each request when it is received. In `handleResponse`, use `console.log` to log the elapsed processing time for each request. Next, add a slow request method to the web worker (e.g., use iteration to compute large Fibonacci numbers). Then, write a loop to send 1000 of these requests to the web worker and examine the logged elapsed request times. What's going on?
3. If the code run by a web worker is CPU-bound, once it starts processing a message, it won't react to further incoming messages until the currently underway computation is done. In this scenario, instead of sending multiple messages to a web worker, we could maintain a queue of requests and only send the worker another when the previous request finishes. Change `createWebWorker` to work in this way.
4. Add to your simple timing statistics to also keep track of how long each request spends in the queue before being sent to the web worker.

5. Enhance your queued version of `createWebWorker` to create a pool of identical web workers. The thread pool size can be passed as an argument. When a request comes in, add it to the request queue and then look for an available worker to process it. Send 1000 messages to the pool of web workers and examine the request times.

Using Web Sockets

Web sockets are a second example of free-form messaging, in which a client and server can send unsolicited messages at any time. There is no built-in formal mechanism for making a call and getting a response.

As in “**Communicating with a Web Worker**” on page 37, we’ll include a request ID each time we send a request, and the server will send it back with the result of the call.

The Web Socket Server

Here’s a node.js server to handle incoming web socket requests:

```
var port = 9999,
    $ = require('jquery-deferred'),
    express = require('express'),
    app = express(),
    server = require('http').createServer(app),
    io = require('socket.io').listen(server);

var METHODS = { // ❶
  add: function(payload){
    return $.Deferred().resolve(payload[0] + payload[1]).promise();
  },
  subtract: function(payload){
    return $.Deferred().resolve(payload[0] - payload[1]).promise();
  },
  divide: function(payload){
    if (payload[1] === 0){ // ❷
      return $.Deferred().reject('Cannot divide by zero.').promise();
    }
    else {
      return $.Deferred().resolve(payload[0] / payload[1]).promise();
    }
  }
};

io.sockets.on('connection', function (socket){
  socket.on('request', function(request){
    var method = request.method,
        func = METHODS[method]; // ❸
    if (func){
      func(request.payload) // ❹
        .done(function(result){
```

```

        socket.emit('response', { // ❸
          payload: result,
          requestId: request.requestId
        });
      })
      .fail(function(error){
        socket.emit('response', { // ❹
          error: 'Calling ' + method + ' failed: ' + error,
          requestId: request.requestId
        });
      })
    }
  );
}
else {
  socket.emit('response', { // ❺
    error: 'Unknown method: ' + method,
    requestId: request.requestId
  });
}
});
});

server.listen(port, '0.0.0.0');

```

- ❶ The METHODS object contains the functions that a client can call and expect replies from. Notice that each of the functions returns a promise that has already been resolved (or rejected). This is just to illustrate that, server side, we expect to be making calls to promise-returning functions. In more typical circumstances, the node server would make calls to other asynchronous methods that returned promises that the called methods would fire themselves.
- ❷ The `div` function checks the denominator of the division, and returns a rejected promise in the case of zero.
- ❸ We look up the method name on all incoming request requests. Note that as in “Communicating with a Web Worker” on page 37, we are not precluding a situation in which the client wants to send other kinds of requests, such as those not requiring a response.
- ❹ Supposing it exists, we call the local (server-side) function, which returns a promise. We use `done` and `fail` callbacks to process its result or error.
- ❺ If the call was successful, we send a response back to the client with the result and also the request ID from the original request. Returning the original request ID allows the client (see below) to route the result to its correct (deferred) destination when it arrives.
- ❻ If the call fails, we send a response back to the client with an error and the request ID from the original request, as above.

- 7 If the client asked us to call a method we don't know about, we send back an error, again including the original request ID.

The Web Socket Client

Below is client code that knows how to send requests and handle responses. It also has an example `main` function that sends off four requests and logs the results. A cleaner design would separate request handling (`makeRequest` and `handleResponse`) from the example usage (`main`).

```
var $ = require('jquery-deferred'),
    socket = require('socket.io-client').connect('http://localhost:9999'),
    requests = {}, // 1
    requestId = 0; // 2

function makeRequest(method, payload){ // 3
    var deferred = requests[requestId] = $.Deferred();
    socket.emit('request', {
        method: method,
        payload: payload,
        requestId: requestId++ // 4
    });
    return deferred.promise();
}

function handleResponse(response){
    var deferred = requests[response.requestId]; // 5
    if (response.error){
        deferred.reject(response.error); // 6
    }
    else {
        deferred.resolve(response.payload); // 7
    }
    delete requests[response.requestId]; // 8
}

function main(){
    var finishedCount = 0, // 9
        possiblyExit = function(){
            if (++finishedCount === 4){
                process.exit();
            }
        };

    makeRequest('add', [20, 15]).then( // 10
        function(result){ console.log('20 + 15 = ', result); },
        function(error){ console.log('Oops!', error); }
    ).always(possiblyExit); // 11

    makeRequest('subtract', [20, 15]).then(
        function(result){ console.log('20 - 15 = ', result); },
```

```

        function(error){ console.log('Oops!', error); }
    ).always(possiblyExit);

    makeRequest('multiply', [20, 15]).then(
        function(result){ console.log('20 * 15 = ', result); },
        function(error){ console.log('Oops!', error); }
    ).always(possiblyExit);

    makeRequest('divide', [20, 0]).then(
        function(result){ console.log('20 / 0 = ', result); },
        function(error){ console.log('Oops!', error); }
    ).always(possiblyExit);
}

socket.on('connect', function(){ // 12
    socket.on('response', handleResponse);
    main();
});

```

- ❶ requests will be keyed by request ID and its values will be outstanding deferreds. These correspond to server requests that have not yet received an answer.
- ❷ requestId will be a unique ID sent in each server request.
- ❸ The makeRequest function does the work of sending a request to the server. It creates a deferred (whose promise will be returned), puts it into requests, and sends a message to the the server.
- ❹ The message to the server includes the request ID, which is then incremented to be ready for sending the next request (if any).
- ❺ When a response arrives, its corresponding deferred is found in requests.
- ❻ If the response contains an error, the original deferred is rejected.
- ❼ If not, the original deferred is resolved with the result from the server.
- ❽ The entry in the requests object is removed, seeing as its deferred has now either been resolved or rejected.
- ❾ In the main client code, we manually keep track of how many of our requests have been answered and exit when all responses are in. See the challenges below for more on this.
- ❿ Each time we call makeRequest with a method and a payload, we use then on the returned promise to set up further processing of the server response.
- ⓫ We add a call to possiblyExit to each promise. When the last call to finish completes, the client will exit. By using always, we make sure possiblyExit is called regardless of whether the server call succeeded or resulted in an error.
- ⓫ Once the web socket connection is made, arrange to process incoming messages using handleResponse.

Running the client code produces the following on the console, as you'd expect:

```
20 + 15 = 35
20 - 15 = 5
Oops! Unknown method: multiply
Oops! Calling divide failed: Cannot divide by zero.
```

Challenges

1. What trivial change could you make to the server code to allow called functions (i.e., those in METHODS) to return results that are not promises?
2. What would happen if the server had a bug and didn't send a request ID in a response, or sent a request ID that the client never sent, or sent a request ID that had already been used? Fix the client code to handle these situations as best you can.
3. In the code above, the client can make a call to the server and get a response, but not vice versa. Make it possible for the server to send a request to the client and get a response.
4. The example client code sends four requests to the server and uses a variable (finishedCount) to count how many responses it has received and to know when to exit. An alternative approach would be to write that part of the client as follows:

```
function main(){
  return $.when(
    makeRequest('add', [20, 15]).then(
      function(result){ console.log('20 + 15 = ', result); },
      function(error){ console.log('Oops!', error); }
    ),
    makeRequest('subtract', [20, 15]).then(
      function(result){ console.log('20 - 15 = ', result); },
      function(error){ console.log('Oops!', error); }
    ),
    makeRequest('multiply', [20, 15]).then(
      function(result){ console.log('20 * 15 = ', result); },
      function(error){ console.log('Oops!', error); }
    ),
    makeRequest('divide', [20, 0]).then(
      function(result){ console.log('20 / 0 = ', result); },
      function(error){ console.log('Oops!', error); }
    )
  );
}

socket.on('connect', function(){
  socket.on('response', handleResponse);
  main().always(function(){
    process.exit();
  });
});
```

This is much more concise. There's only one small problem: it won't work! Why not?

5. Read the jQuery source code and find the line that causes the above `$.when` approach to fail. How could you change `$.when` to not behave that way, and what would that mean for code that used it? Given that the signature of `$.when` doesn't permit backward-compatible changes, how could you write a more flexible `$.when`? How could the call signature of `$.when` have been written to allow future changes so we wouldn't be stuck with this behavior?

Automatically Retrying Failing Deferred Calls

These days a common need is to write code that talks to services that may be briefly unavailable. If an error of some kind occurs, the correct (and documented) action to take is just to retry the original call a little later. Examples include the APIs for Amazon's S3 service and Twitter. Transient API failures occur fairly frequently with both those services.

Below is a function where you pass a context (i.e., a value for `this`), a function to be called, and some arguments for the function. If the function call fails, it is retried repeatedly after increasing delays. The original failure value is returned (via a promise that is rejected) if all attempts to call the function fail:

```
function retryingCaller( /* context, function, args... */ ){
    var context = arguments[0],
        func = arguments[1],
        args = arguments.slice(2),
        deferred = $.Deferred(),
        rejectValue,
        delays = [0.0, 0.01, 0.02, 0.05, 0.1, 0.15, 0.2, 1.0, 2.0],
        attempt = 0,

    wait = function(timeout){ // ❶
        var d = $.Deferred();
        setTimeout(d.resolve, timeout);
        return d.promise();
    },

    error = function(value){ // ❷
        if (rejectValue === undefined){
            rejectValue = value; // ❸
        }
        if (attempt === delays.length){
            deferred.reject(rejectValue); // ❹
        }
        else {
            call(); // ❺
        }
    }
}
```

```

    },
    call = function(){
      wait(delays[attempt++]).done( // ❹
        function(){
          $.when(func.apply(context, args)).then(
            deferred.resolve, // ❺
            error // ❻
          );
        }
      );
    };

    call();
    return deferred.promise();
  }
}

```

- ❶ This is the wait function we wrote earlier, in “A Replacement for the `setTimeout` Function” on page 21.
- ❷ error will be run each time a promise returned by the underlying function is rejected. Its job is to save the first reject value and to either reject the master deferred (whose promise is returned by `retryingCaller`) or to arrange for the underlying function to be called again.
- ❸ Save the first reject value (see the challenges below for different approaches to rejecting the master deferred).
- ❹ If the number of allowed attempts has been reached, reject the master deferred with the original reject value.
- ❺ We still have some attempts left, so call the underlying function again.
- ❻ Wait for the next delay time to elapse before calling the underlying function.
- ❼ If the underlying function runs without error, pass its return value along to the master deferred.
- ❽ If the underlying function hits an error, call our error function, which will either arrange for a retry or reject the master deferred.

Challenges

1. Make it possible for the caller to specify the delay intervals.
2. If all calls to the passed function fail, the promise returned by `retryingCaller` should be rejected with the error value from the *first* failing call. But there’s one condition under which this is violated. Figure out what can go wrong and fix the bug.
3. Instead of rejecting the deferred with the original error value, reject it with the final value, or keep a list of the failure values and reject it with a list of all failures.

4. Our code is circular: the `call` function arranges to invoke `error` if something goes wrong, but `error` invokes `call` if the maximum number of attempts has not been reached. Is there a way this circularity could cause the stack to fill with invocations of `call`, `error`, and the function passed to `retryingCaller`?
5. Make it possible for the caller to pass an error-checking function. Each time the underlying function fails, pass the error value to the checking function. If it returns `true`, continue to retry the underlying function; otherwise, reject the master deferred. This allows a caller to provide an error-checking function that, for example, could cause `retryingCaller` to retry the function on HTTP 500 (Internal Server Error), 502 (Bad Gateway), and 504 (Gateway Timeout) errors, but to not retry on any other HTTP error.

As another possibility, an error checker could allow just two 404 (Not Found) error retries, as illustrated below:

```
function tester(){  
  var count = 0, max404errors = 2;  
  return function(error){  
    return error.status === 404 ? count++ < max404errors : true;  
  };  
}
```

Memoization

Memoization is a simple technique for greatly speeding up repeat calls to an **idempotent function**. Informally, an idempotent function is one that returns the same value each time you call it with the same arguments. In those cases, it is safe to store (“memoize”) the function’s result the first time it is called and to return the known result thereafter for identical calls. Ignoring issues of the memory used to store results, memoization of computationally expensive functions has the happy result of delivering massive speed gains.

Simple memoization is easy to implement in many programming languages, including JavaScript. At first glance, though, it seems like memoization in the context of deferreds might be more involved. There are three cases we need to consider:

- a. The function to be memoized has never before been called with the given argument.
- b. The function has already been called with this argument, but the promise the memoizing function originally returned has not yet been resolved.
- c. The function has already been called with this argument, and the promise that was originally returned has been resolved.

In addition, if we want to be able to memoize arbitrary functions, we'll have to deal with functions that return promises, functions that return nonpromises, and functions that can return either.

Fortunately, it turns out that this is one of the rare cases where a very simple solution solves what seems like a tricky situation:

```
function memoize(func){  
  var promises = {}; // ❶  
  
  return function(arg){  
    if (promises.hasOwnProperty(arg) === false){ // ❷  
      promises[arg] = $.when(func(arg)).promise(); // ❸  
    }  
    return promises[arg]; // ❹  
  };  
}
```

- ❶ The promises object holds promises corresponding to calls already made to func.
- ❷ When a call is made, check for an existing promise for the given argument.
- ❸ If the promise doesn't exist, this is the first time we've been called with this argument. Call func on the argument, use \$.when to create a promise that will be resolved with the result, and store the new promise.
- ❹ Return the promise for this call.

Discussion

Firstly, we've made a simplifying assumption: that func will not be called with JavaScript objects (lists are objects too) as an argument. See the challenges below if this makes you feel better.

If you've encountered memoization before, the code above should seem completely unremarkable. But, as simple as it seems, this example deserves some discussion. How does this simple function handle all three cases listed above?

Firstly, it turns out that cases (b) and (c) above can be treated identically. One of the beauties of deferreds is that you can use them in blissful ignorance of whether they have been resolved yet or not. In the case of (b), we have an unresolved promise in the promises object. It has already been returned to at least one caller. If another call is made with the same argument, the same unresolved promise is again returned. When the result arrives, both callers will get it. In the case of (c), the promise corresponding to the passed argument has already fired, but we're still holding the promise and we can simply return it again. The promise already contains the result, which will be passed immediately to any additional callbacks added to it by our caller.

In normal memoization, a cache of known results is maintained. In our case, we keep a cache of promises. Because we wrap the call to `func` inside `$.when`, the function returned by `memoize` will *always* return a promise, even if `func` does not. This is a departure from strict memoization. To repeat, the result from `func` is not returned: instead, we return a promise that will resolve with that result. It is the use of `$.when` that allows us to ignore the issue of whether `func` returns a promise or not. Because `$.when` handles both cases, we don't need to check the returned value.

Avoiding the Dogpile Effect

Let's compare deferred-based memoization to memoization without deferreds.

Without deferreds, what happens if a first call comes in with an argument `X`, but before `func(X)` has finished computing there is another call with argument `X`? Then another, and another, and maybe thousands of others? The nondeferred code winds up calling `func(X)` many times. That is, it does exactly the thing we are trying to avoid in the first place! This is informally known as the “dogpile effect.” It can result in all sorts of trouble, including memory exhaustion, CPU overload, or a form of **denial of service attack**.

With deferreds though, we immediately have a placeholder for the result of the first call (with any given argument) to the memoized function. With that promise safely stored, we can just return it to the hordes of callers that hit the service before the result is in. When the result finally arrives and the deferred for that call is resolved, all the callers will receive it. Deferreds solve this problem so cleanly, without you having to lift a finger, that you may not have even realized there was a potential problem there at all. Neat.

This effortless avoidance of the dogpile effect is a great example of the value of deferreds.

Challenges

1. Modify `memoize` to allow a context argument to be passed as well as a function. The context should be used as `this` when `func` is called.
2. What happens if `func` is called with a JavaScript object?
3. How would you modify the above code if `func` took two arguments instead of one? What if it took any number of arguments, each of arbitrary type?
4. The returned function will give the same promise to multiple callers. Will this create a problem if callers each add their own callbacks to the promise?
5. How would you change `memoize` if you wanted to be stricter and only return a promise when `func` did?
6. What would be the effect of not wrapping the `func(arg)` call in `$.when`?

7. The `promises` object can only grow in size. It would be good to discard results that have not been accessed recently. Add a housekeeping function to `memoize` that's invoked periodically by `setInterval` to delete items from `promises` that haven't been accessed recently.

Short-Term Memoization of In-Progress Function Calls

Regular functions compute their results each time they're called. At the other extreme, memoized functions can remember earlier results forever and never recompute anything. In this example we'll consider the middle ground, functions that are only briefly memoized.

An application built by the authors consisted of multiple front-end HTTP servers using RPC to internal servers to fulfill API requests. One HTTP API request was called `createUser`. In a typical setup, a request to create a user in a system will either get a successful result upon user creation or an error if the user already exists. But it is also possible that while a request to create a specific user is underway, another request arrives to create the same username. A simplistic outcome in this situation might be that the first request receives a successful status code and the second results in a database transaction error that is translated into a `UserExists` error.

We can easily improve on this, though, if we're using deferreds to create users. When a request arrives to create a user and a promise-returning function is called to do the work, we can keep track of the promise as well as returning it to our caller. Then, if a second call arrives to create the same user before the first has finished, we simply return the same promise. At that point, our code will have returned the same promise twice and our callers will both receive the same result. Once the original request has finished, we can forget the promise and allow subsequent requests to flow through the system as normal. Here's the code:

```
var promises = {}; // ❶

function createUser(username){
  if (promises.hasOwnProperty(username)){ // ❷
    return promises[username];
  }
  else {
    var promise = internalCreateUser(username); // ❸
    promises[username] = promise; // ❹
    promise.always(function(){
      delete promises[username]; // ❺
    });
    return promise;
  }
}
```

- ❶ The `promises` object holds promises corresponding to in-progress `createUser` requests. This is exactly as in “Memoization” on page 49.
- ❷ `createUser` first checks to see if `promises` contains a promise for the given username, and if so it simply returns it. This is the case where a second (or third, etc.) request for an identical username arrives while a first request is in progress.
- ❸ If there is no call in progress, a call is made to the internal function that does the work of making a user. This function is assumed to return a promise.
- ❹ The promise from `internalCreateUser` is stored in the `promises` object.
- ❺ A callback is added to the promise returned by `internalCreateUser` that will remove the promise from `promises`. As a result, subsequent calls to `createUser` will trigger another call to `internalCreateUser`, even if the call has been made before.

createUser Is Not Idempotent

In regular memoization, the function being memoized is idempotent: the result of a function call is assumed to never change and can be remembered for all time. But `createUser` is not idempotent. When it is called with a nonexistent username, it returns a value to indicate the user was created. On subsequent calls (with the same username), it returns a value to indicate the user already existed. So we’re not doing regular (permanent) memoizing of the `createUser` result. We’re just making sure that if there’s an in-progress call for a given argument, we use its result instead of making a second simultaneous and identical call to `internalCreateUser`. The memoization is short-term; it only lasts as long as the function call is outstanding. It’s a funny kind of memoization, because we’re not storing the result of the function call, we’re effectively storing the in-progress function call. We return to this discussion in “Promises Are First-Class Objects for Function Calls” on page 79.

Challenges

1. What undesirable outcome could occur if the promise from `internalCreateUser` were put into the `promises` object *after* the always callback is set up?
2. The problem in the previous challenge is an example that illustrates a general principle of maintaining data structures that hold deferreds and promises. Can you generalize our specific problem to arrive at the principle?
3. Think about the similarities and differences between the short-term in-progress function call memoization above and regular caching with result expiry after a fixed time.

Streaming Promise Events

In this example we'll do some curious things in order to build up a tool we'll put to good use in a later example.

Take a look at the following function and figure out what it does (and doesn't do!) before reading on:

```
function simpleEventStream(promises){
  var i, deferred = $.Deferred(),
      stream = function(){
        deferred.notify.apply(deferred, arguments);
      };

  for (i = 0; i < promises.length; i++){
    promises[i].done(stream).fail(stream).progress(stream);
  }

  return deferred.promise();
}
```

There are a couple of odd things going on here. The function accepts a list of promises and it adds the same `stream` function to all of the `done`, `fail`, and `progress` callback lists. The `stream` function calls the `notify` function on the deferred whose promise is returned by `simpleEventStream`. So, every time one of the passed promises is resolved, rejected, or notified of progress, the arguments of that call will be passed on as progress events on the returned promise. In other words, `simpleEventStream` takes all events generated by a given set of promises and delivers them, as they occur, to its caller via a stream of progress events on its returned promise.

But there's something else that's odd about `simpleEventStream`. It never calls `resolve` or `reject` on the deferred it creates! Hmm...shouldn't that be illegal, or something? Not really. There's no rule that says a deferred has to be fired. Of course they usually are, but we have a case where we're using a deferred for something slightly different than a one-time conclusion.

Delivering More Information

There's an obvious first improvement we could make to the event streamer: report the index of the promise that created the event as well as whether the promise was resolved, rejected, or notified. That's an easy addition:

```
function eventStream(promises){
  var i, deferred = $.Deferred(),
      callNotify = function(index, type){
        return function(){
          deferred.notify(index, type, arguments);
        };
      };

  for (i = 0; i < promises.length; i++){
    promises[i].done(callNotify(i, 'resolved')).fail(callNotify(i, 'rejected')).progress(callNotify(i, 'progress'));
  }

  return deferred.promise();
}
```

```

    for (i = 0; i < promises.length; i++){
      promises[i]
        .done(callNotify(i, 'resolve'))
        .fail(callNotify(i, 'reject'))
        .progress(callNotify(i, 'notify'));
    }

    return deferred.promise();
  }
}

```

Here, any progress callback added to the promise returned by `eventStream` will be called with the index of the promise that created the event, the type of event (resolution, rejection, or progress notification), and the arguments provided to the promise.

If you're wondering what use this could possibly be, imagine you had a set of promises and you wanted to log all their events, in order. You could of course manually add `done`, `fail`, and `progress` logging functions to each promise and then collate the output, but it would be simpler to just pass the promises to `eventStream` and add a single progress function to do the logging.

That may not seem particularly useful, but we're not done yet.

Delegating the Event Stream

The `eventStream` function is interesting, but so far it doesn't seem all that useful unless you want to observe promise events. The promise we received from `eventStream` never fires. What if we wanted it to be resolved or rejected under certain circumstances?

Here's a slight variation on the above, `delegateEventStream`, which also returns a promise:

```

function delegateEventStream(promises, eventHandler){ // ❶
  var i, deferred = $.Deferred(),
      callHandler = function(index, type){ // ❷
        return function(){
          eventHandler(index, type, arguments, deferred); // ❸
        };
      };

  for (i = 0; i < promises.length; i++){
    promises[i]
      .done(callHandler(i, 'resolve'))
      .fail(callHandler(i, 'reject'))
      .handler(callHandler(i, 'notify'));
  }

  return deferred.promise();
}

```

A little explanation is probably in order:

- ❶ `delegateEventStream` must be passed an array of promises and an event handler function.
- ❷ The `callHandler` function returns a function that will call the event handler.
- ❸ Each time the event handler function is called, it is passed the index of the promise, the type of event ('resolve', 'reject', or 'notify'), the arguments involved, and the deferred `delegateEventStream` created.

As you may have noticed, although `delegateEventStream` creates a deferred, it not only never resolves it or rejects it, it also never even calls `notify` on it as the earlier examples did. Even odder, it passes the deferred off to an unspecified event handler function.

What's going on?

To Be Continued...

As unsatisfying as it might be, we're going to leave this little experiment for now. You can take solace from the variety of unusual tricks we used in working towards `delegateEventStream`. We made a deferred that was never resolved or rejected, we used a progress callback to deliver events from other deferreds, and, finally, we created a deferred that we never used, but which we instead pass to another function that we didn't even write.

Stay tuned to see how `delegateEventStream` saves the day in a later example.

Challenges

1. What would change if `simpleEventStream` used `promises[i].then(stream, stream, stream)` instead of chaining calls to `done`, `fail`, and `progress`?
2. Modify `eventStream` so that it keeps a count of the number of its promises that have resolved or been rejected and calls `resolve` on the deferred whose promise it returned when all promises have fired. This is like a final signal to the caller of `eventStream` to indicate that there will be no further events.
3. Prove you understand `delegateEventStream` by writing a function called `clonePromise` that takes a promise and uses `delegateEventStream` to return another one that acts in the same way. We can't think of any reason why you'd want to do that, but don't let that get in the way of stretching your mind!

Getting the First Result from a Set of Promises

There are occasions when you have multiple ways of obtaining a needed value. You don't care which one is used, you just want the result as quickly as possible. For example, server-side code might need to retrieve a user's avatar image. The image might be stored in a fast cache, such as **Redis** or **memcached**, it might be stored in a distributed filesystem, or you may need to fall back to making an external network request (perhaps to **Gravatar**) to get an image after retrieving the user's email address from either a cache or a database.

The normal approach is to query the potential sources in order of expected response time. You look first in the cache. If that fails, you check the local filesystem. If that fails, you make a network request. The problem is that every time the best case doesn't happen, the call that is finally successful doesn't even start until after the earlier attempts have all failed.

An improved approach (at least if speed is all you care about) is to launch the different attempts all at once. Then you take the first result that comes back. Managing this kind of processing seems pretty easy if you're working with promises. Let's start with the simplest approach:

```
function fastestPromise(promises){
  var deferred = $.Deferred(),
      done = function(result){
        deferred.resolve(result);
      },
      fail = function(error){
        deferred.reject(error);
      };

  for (var i = 0; i < promises.length; i++){
    promises[i].done(done).fail(fail);
  }

  return deferred.promise();
}
```

If you've been reading your way through this book, the above code will hopefully need no explanation. If you need a summary, though, the `fastestPromise` function adds a `done` and a `fail` callback to all the promises you pass it. The callbacks simply take the value they're passed and use it to fire the deferred whose promise `fastestPromise` originally returned. Because it doesn't matter how many times a deferred is fired, we don't have to worry about subsequent promises finishing and also trying to fire the promise `fastestPromise` returned. Only the first call to `resolve` or `reject` counts; the rest have no effect.

Which Promise Fired?

This simplistic function can be made more useful. Our caller will likely want to know which of their promises was the first to fire. That way, if the response doesn't come from the cache, they can cache it. Or if the result comes in while network requests are still ongoing, it may be possible to cancel them.

So let's pass the index of the promise that fires first back to the caller:

```
function fastestPromiseWithIndex(promises){
  var deferred = $.Deferred(),
      makeDone = function(index){
        return function(result){
          deferred.resolve(index, result);
        };
      },
      makeFail = function(index){
        return function(error){
          deferred.reject(index, error);
        };
      };

  for (var i = 0; i < promises.length; i++){
    promises[i].done(makeDone(i)).fail(makeFail(i));
  }

  return deferred.promise();
}
```

We can use the function as follows, supposing `avatarFromRedis`, `avatarFromFilesystem` and `avatarFromGravatar` have all been defined elsewhere and that each of them returns a promise:

```
var fastest = fastestPromiseWithIndex([
  avatarFromRedis('joe'),
  avatarFromFilesystem('joe'),
  avatarFromGravatar('joe@example.com')
])
.then(function(index, avatar){
  if (index !== 0){
    // The response was not from Redis. Add the avatar info to Redis so
    // we have it cached.
  }

  if (index !== 2){
    // The response was not from Gravatar. Cancel the outstanding
    // Gravatar network request.
  }

  return avatar;
});
```

```
fastest.done(function(avatar){  
  // Use the avatar.  
});
```

Note that the promise returned by `fastestPromiseWithIndex` fires with two arguments: the index of the promise arguments that fired and the result (or error). The `then` function examines the returned `index` and updates the Redis cache or cancels the network request, as appropriate, and passes along just the `avatar` result.

A Fly in the Soup

It seems like we're done, but in fact the reasoning behind the above approach is massively flawed. Can you figure out what we've overlooked? (Hint: What do you think `avatarFromRedis` might do if it doesn't find a key?)

The problem has to do with timing, cache misses, and failure handling. The `avatarFromRedis` call is probably always going to be the first to complete. What should it do if it cannot find the key in question? Rejecting its deferred will cause `fastestPromiseWithIndex` to in turn reject its deferred, which is not what we want at all. If `avatarFromRedis` instead resolves its deferred with a value to indicate a missing key, `fastestPromiseWithIndex` will resolve its deferred and pass along the nonanswer. That's no use to us either. We can't really expect `avatarFromRedis` to simply be quiet if it doesn't find a key. In any case, it should be able to reject its deferred if it runs into some other error. It might even be that the Redis service has been stopped or is unreachable for some reason. That shouldn't prevent us from getting an answer from another source, though. We have exactly the same set of problems with `avatarFromFileSystem`, which might also not find the file it's asked for, etc.

What to do?

We don't want to revert to calling our functions sequentially, so we need to figure out a way to ignore cache misses, and maybe other errors, that come from `avatarFromRedis` and `avatarFromFileSystem` (and `avatarFromGravatar` too, for that matter). We need more control over the promise we get back from `fastestPromiseWithIndex`.

delegateEventStream Redux

If you managed to stay awake all the way through the last example ("[Streaming Promise Events](#)" on [page 54](#)), you might realize that the `delegateEventStream` function we built is just what we need now.

All we need is a handler function for the events coming from the promises returned by `avatarFromRedis` and friends. For simplicity, let's assume that those functions all call `resolve` on the deferreds they create when they can't find what they're asked for and

that in that case they send a JavaScript object with a `status` attribute set to 404. Here's the event handler we need:

```
function eventHandler(index, type, args, deferred){
  if (type === 'resolve'){
    if (args[0].status === 404){
      // Ignore any not-found errors.
    }
    else {
      deferred.resolve.apply(deferred, args);
    }
  }
  else if (type === 'reject'){
    deferred.reject.apply(deferred, args);
  }
}
```

Using `eventHandler` we arrive at a solution that looks just like our earlier one that used `fastestPromiseWithIndex`, but which instead uses `delegateEventStream`:

```
var fastest = delegateEventStream([avatarFromRedis('joe'),
                                  avatarFromFilesystem('joe'),
                                  avatarFromGravatar('joe@example.com')],
                                  eventHandler);

fastest.done(function(avatar){
  // Use the avatar.
});
```

Challenges

1. Add error processing to the code above that uses `fastestPromiseWithIndex`.
2. What will happen if `fastestPromise` or `fastestPromiseWithIndex` is called with an empty list of promises?
3. In the code that uses `fastestPromiseWithIndex`, the user's email is conveniently embedded in the JavaScript. This is of course unrealistic. Modify the code so that it looks in both Redis and a database to find the email address from the username, passing the result to `avatarFromGravatar`. You can assume the existence of promise-returning functions `emailFromRedis` and `emailFromDatabase`. Of course you should use `fastestPromiseWithIndex` to initially get the email address the fastest way possible, and add it to Redis if it is not already there. For extra credit, replace the use of `fastestPromiseWithIndex` with `delegateEventStream`.
4. A nice feature about `$.when` is that you can safely pass it objects that are not promises. It considers any such argument to be, essentially, a value that has already been returned by a promise. Modify `fastestPromiseWithIndex` to behave in the same way—i.e., if any argument is not a promise, resolve the deferred you created (whose

promise you're about to return). This adds flexibility because the three avatar-fetching functions will no longer need to return promises. You could, for example, keep an in-memory cache of the most recent user avatars and pass `fastestPromiseWithIndex` a function that returns a synchronous result from that cache. Hint: jQuery heuristically decides if an argument to `$.when` is a promise by checking to see if it has a `promise` attribute that's a function.

5. How would you change `eventHandler` to ignore failures due to Redis being unreachable?
6. How would you change `eventHandler` if one of the called lookup functions rejected its deferred whenever it couldn't find something (instead of resolving it with a 404 status)? Keep in mind that a lookup function might reject its deferred for other reasons too.
7. In the solution using `delegateEventStream`, we don't populate Redis or the file-system in case of lookup misses. How would you fix that?
8. There is a small but important bug in our final solution. What is it and how can you fix it?

A Deferred Queue

The Python event-driven networking engine **Twisted** is entirely based on deferreds. **Twisted deferreds** are conceptually identical to those in jQuery, but the details of their behavior differs in several important ways.

An extraordinarily elegant small class in Twisted is **DeferredQueue**.

Before we examine it, though, think about code for a regular synchronous queue. What happens if you call `get` on a regular queue that's empty? Almost certainly one of two things: you'll get some kind of `QueueEmpty` error, or else your code will block until some other code puts something into the queue. That is, you either get a synchronous error or you get a synchronous nonempty response.

Below is a simplified version of the Twisted class, written for use in node.js. The `makeQueue` function returns an object that has `queue` `get` and `put` methods on it.

```
function makeQueue(){
  var $ = require('jquery-deferred'),
      waiting = [], // ❶
      queue = []; // ❷

  return {
    get: function(){
      var deferred;
      if (queue.length){
        deferred = $.Deferred().resolve(queue.shift()); // ❸
      }
    }
  }
}
```

```

    }
    else {
        deferred = $.Deferred();
        waiting.push(deferred); // ❹
    }
    return deferred.promise();
},

put: function(item){
    if (waiting.length){
        waiting.shift().resolve(item); // ❺
    }
    else {
        queue.push(item); // ❻
    }
}
};
}

```

- ❶ `waiting` is a list of unresolved deferreds whose promises were returned to callers of `get` when there were no items in the queue. As items are put into the queue, these deferreds will be resolved.
- ❷ `queue` is a list of items that have been put, but which have not yet been retrieved with a `get`.
- ❸ When a call is made to `get` and there are items in the queue, a new deferred is made, the first item in the queue is used to resolve it immediately, and its promise is returned to the caller. Any callbacks the caller adds to the returned promise will run at once because the deferred has already been resolved.
- ❹ If there is nothing in the queue in `get`, a new deferred is made, it is added to `waiting` (so we can later resolve it when an item arrives via `put`), and its promise is returned.
- ❺ When `put` is called, if an earlier caller to `get` received an unresolved promise (i.e., the queue was empty in `get` so it returned a promise whose deferred is still in `waiting`), resolve its deferred with the incoming item.
- ❻ If there are no earlier callers who have not yet received a result, push the new item onto the queue so it can be given to a future caller of `get`.

The Twisted DeferredQueue class is a thing of beauty. It's a valuable example. This kind of coding with deferreds seems routine and straightforward to us now. But it certainly wasn't always that way. If you don't 100% understand the above, we highly recommend that you stop and take the time to read it again and to think about it. Once you really "get" it, you'll be well on your way to mastering deferreds.

Challenges

1. Implement a deferred priority queue. The `put` function should accept an optional numeric priority argument and `get` should return a promise that fires with the highest priority element.
2. Enhance the priority queue with functions that allow an element to be deleted from the queue or to be re-prioritized.

when2: An Improved jQuery.when

In “[Getting the First Result from a Set of Promises](#)” on page 57, we focused on requesting the same information (a user’s avatar) from various possible sources and acting on the first response. While our solution is useful, the implementation was a bit simplistic and not as general as it could have been.

An obvious first generalization is that we may want to make several calls that are not designed to return the same information. For example, we might make a web socket API request to a remote server and want to set a timeout on the reply. If we have a deferred for both the web socket request and the timeout, it would be useful to have a way to get a third deferred that resolves when the first of the other two is resolved.

These situations have a lot in common with what `$.when` provides. `$.when` returns a promise that resolves when *all* its arguments are resolved. In “[Getting the First Result from a Set of Promises](#)” on page 57, we wanted a promise that resolves as soon as *any* of its promises is resolved.

There’s another aspect of `$.when` that can be limiting: as soon as any of its promises is rejected, it rejects the deferred whose promise it returned. So it doesn’t provide a mechanism for monitoring a collection of promises and getting results from all of them—including any errors. We’ve actually already run into this limitation, in the web socket “[Challenges](#)” on page 46.

In summary, there are three common actions that might be wanted on a set of promises:

1. Resolve on the first success.
2. Reject on the first error (the `$.when` behavior).
3. Resolve when all results (successes or errors) have been collected.

These behaviors don’t seem that different. Given their similarity to `$.when`, it shouldn’t be that hard to modify `$.when` to make it more flexible.

We wrote `when2` to do exactly that. You can load it after loading jQuery (hence its use of jQuery instead of `$`). Our `when2` function is based on the `$.when` jQuery *deferred.js* source. The source for `when2`, and a test suite, can also be found [on GitHub](#).

The `when2` function takes two arguments: `promises`, which is a list of promises (that may include nonpromise values) and `options`, an optional JavaScript object. The `options` object lets us choose among the three behaviors just described:

1. If `options.resolveOnFirstSuccess` is true, the promise returned by `when2` will resolve as soon as any of the passed deferreds resolves. The `always` and `done` callbacks will be passed `index` and `value` arguments, where `index` is the index of the deferred that fired. If nonpromises are in the arguments to `when2`, the first one seen will trigger the `resolve` callbacks (not very useful, but consistent).
2. If `options.rejectOnFirstError` is false, the promise returned by `when2` will never be rejected. It will collect the results (successes and failures) from all the passed promises and deliver them all at once via `resolve`. The `when2` caller gets to figure out which values, if any, are errors.
3. If no `options` argument is passed or `options.rejectOnFirstError` is true, the promise returned by `when2` will be rejected on the first error. This is the behavior of `$.when`. The arguments passed to `always` and `fail` callbacks will be an `index` and a `value`.

```
jQuery.when2 = function(promises, options) { // ❶
    var i,
        coreSlice = [].slice,
        resolveValues = promises ? coreSlice.call(promises) : [], // ❷
        length = resolveValues.length,
        resolveContexts = length ? new Array(length) : window,
        remaining = length, // ❸
        promiseArgFound = false, // ❹
        resolveOnFirstSuccess = (options &&
            options.resolveOnFirstSuccess), // ❺
        rejectOnFirstError = !options || options.rejectOnFirstError, // ❻

        deferred = jQuery.Deferred(), // ❼

        doneFunc = function(i) { // ❽
            return function(value) {
                if (deferred.state() === 'pending') {
                    --remaining; // ❾
                    resolveContexts[i] = this;
                    resolveValues[i] = arguments.length > 1 ?
                        coreSlice.call(arguments) : value;
                    if (resolveOnFirstSuccess) {
                        deferred.resolveWith(
                            this, [i, resolveValues[i]]); // ❿
                    }
                }
            };
        };
    for (i = 0; i < length; i++) {
        doneFunc(i).call(promises[i]);
    }
    return deferred;
};
```



```

    }
    else if (remaining === 0){ // 11
        deferred.resolveWith(resolveContexts, resolveValues);
    }
}
};
},

failFunc = function(i){ // 12
    if (rejectOnFirstError){
        return function(value){
            if (deferred.state() === 'pending'){
                value = arguments.length > 1 ?
                    coreSlice.call(arguments) : value;
                deferred.rejectWith(this, [i, value]); // 13
            }
        };
    }
    else {
        return function(value){
            if (deferred.state() === 'pending'){
                resolveContexts[i] = this;
                resolveValues[i] = arguments.length > 1 ?
                    coreSlice.call(arguments) : value; // 14
            }
            if (!--remaining){ // 15
                deferred.resolveWith(
                    resolveContexts, resolveValues);
            }
        };
    }
};

progressFunc = function(i){ // 16
    return function(value){
        if (deferred.state() === 'pending'){
            value = arguments.length > 1 ?
                coreSlice.call(arguments) : value;
            deferred.notifyWith(this, [i, value]);
        }
    };
};

for (i = 0; i < length; i++){ // 17
    if (resolveValues[i] && // 18
        jQuery.isFunction(resolveValues[i].promise)){
        promiseArgFound = true;
        resolveValues[i].promise()
            .done(doneFunc(i))
            .fail(failFunc(i))
            .progress(progressFunc(i));
    }
}

```

```

    else {
      if (resolveOnFirstSuccess){ // 19
        deferred.resolveWith(window, [i, resolveValues[i]]);
      }
      resolveContexts[i] = window;
      --remaining; // 20
    }
  }

  if (promiseArgFound === false){
    deferred.resolveWith(resolveContexts, resolveValues); // 21
  }

  return deferred.promise();
};

```

There's quite a bit of detail (inherited from `$.when`) in the above that we won't try to explain, seeing as the main point here is to understand deferreds rather than all the idiosyncracies of JavaScript. Even ignoring those details, there's a lot going on in `when2`. Take a deep breath.

Here's how it works:

- ❶ Accept two arguments: a list of promises (this may include nonpromise values) and an optional JavaScript object to indicate when the returned promise should be resolved or rejected, as described above.
- ❷ `resolveValues` will collect the results (and errors, if `rejectOnFirstError` is false) returned by the passed promises.
- ❸ `remaining` will be used to count the number of outstanding promises yet to fire. When it reaches zero, the promise returned by `when2` will be resolved (assuming it has not already been resolved or rejected due to the passed value of `resolveOnFirstSuccess` or `rejectOnFirstError`).
- ❹ `promiseArgFound` tracks whether any of the arguments passed to `when2` have promise methods. If no promises are passed to `when2`, the promise it returns will already have been fired (since there is nothing to wait for).
- ❺ `resolveOnFirstSuccess`, if true, will cause the returned promise to resolve as soon as any of the passed arguments is resolved. Note that any passed argument that does not have a promise method will be counted as already being “resolved,” and will trigger a resolve call on the deferred whose promise is returned.
- ❻ `rejectOnFirstError`, if true (or if no options object is passed), will cause the returned promise to be rejected as soon as any of the passed arguments is rejected. This is the behavior of `$.when`.
- ❼ Create the master deferred, whose promise will be returned.

- ❶ `doneFunc` returns a callback function that can be added to the `done` list of a passed promise.
- ❷ One of the passed promises has resolved, so decrement the count of remaining outstanding promises.
- ❸ If `resolveOnFirstSuccess` was `true`, we immediately resolve the master deferred. Instead of just passing the result along with no indication of *which* promise fired, pass the index of the promise as well.
- ❹ Otherwise, if the count of remaining promises to fire is now zero, resolve the master deferred.
- ❺ `failFunc` is similar to `doneFunc`, but it handles the case where a passed promise is rejected.
- ❻ If `rejectOnFirstError` is `true`, we reject the master deferred. Again, we pass the index of the promise that was rejected along with the error.
- ❼ If `rejectOnFirstError` is not `true`, the error value is stored in the `resolveValues` array (whose name is therefore a slight misnomer).
- ❽ If there are no outstanding promises remaining, resolve the master deferred with the accumulated results.
- ❾ `progressFunc` mirrors `doneFunc` and `failFunc`, though it is simpler. Note that progress callbacks will also be called with an index and a value.
- ❿ Loop over the passed arguments.
- ⓫ If an argument has a `promise` method, we assume it's something we can add `done`, `fail`, and `progress` callbacks to, and do so. We also set `promiseArgFound` to `true` so we'll know that the returned promise should not be fired immediately.
- ⓬ If an argument is not a promise and `resolveOnFirstSuccess` is `true`, resolve the master deferred. This is probably not very useful, but it's consistent with the treatment of these values as, essentially, already resolved promises by `$.when`.
- ⓭ The count of outstanding promises is decremented if an argument is not a promise.
- ⓮ Finally, if no promise argument was found, resolve the master deferred with all the passed arguments.

Using `when2` to Time Out a Single Promise

We can use the `resolveOnFirstSuccess` option to `when2` in a nice way that lets us time out a promise.

The trick is to call `when2` with the promise you want to time out and a promise returned from the `wait` function we wrote earlier (in “A Replacement for the `setTimeout` Function” on page 21), and set the `resolveOnFirstSuccess` option to `true`. `when2` returns a new promise that will fire either because the `wait` promise has resolved or because your original promise did.

To be more concrete, suppose you have a variable called `promise` returned by some other function and you want to allow it 200 milliseconds to fire before taking some other action. You’d write:

```
$.when2(
  [wait(200), promise],
  { resolveOnFirstSuccess: true })
.done(function(index, result){
  if (index === 0){
    // Timeout!
  }
  else {
    // The promise was resolved with the given result.
  }
});
```

This small example again illustrates the building-block beauty of deferreds. If you don’t have quite what you need, make a new deferred and hook your old one up to it to give you the behavior you need.

Differences from `$.when`

Here’s a summary of the key differences from the `$.when` function:

- You must pass a list of promises to `when2` instead of just passing all your promises as individual arguments.
- You can optionally also pass a JavaScript object with `resolveOnFirstSuccess` and/or `rejectOnFirstError` attributes. The default behavior is to fail on the first error, just like `$.when`.
- When either (or both) of `resolveOnFirstSuccess` or `rejectOnFirstError` is in effect, all callbacks are called with two arguments: an index into the passed list of promises (so you can figure out which promise fired) and the result.
- progress callbacks added to a promise returned by `when2` will always be called with an index and a result, as above.
- If `rejectOnFirstError` is false, the `when2` master deferred will never be rejected. It will be resolved with the result of all promises, each with a success or error value (you get to figure out which is which).

Challenges

1. How would the behavior of `when2` change if the line with the `resolveWith` call at the end of the function was commented out?
2. What would change if the `if (deferred.state() === 'pending')` guard tests were all removed?
3. An alternative behavior when a nonpromise argument is passed and `resolveOnFirstSuccess` is true is to just store the argument in `resolveValues` and not resolve the master deferred until an actual promise argument (if any) is resolved. How would you make this simple change? What should happen in the case that no promises are passed to `when2`?
4. We saw a neat way to time out a single promise using `when2`. Unfortunately, that trick won't work if we try to time out more than one promise. Why not?
5. Imagine that a caller of `when2` wasn't interested in receiving progress updates from the passed promises but instead wanted to know when each passed promise was resolved. For example, you might send `write` commands to three mirrored servers and you want to know when any two of them have completed the write. Modify `when2` so that each time one of the passed promises is resolved, the index and the result is passed to the progress callbacks. Note that the promise returned by `when2` will still be resolved or rejected exactly as above; you're just repurposing the progress events. You can add to the `options` argument to let the caller request this behavior. See also [“Streaming Promise Events” on page 54](#).

Timing Out Promises

In [“when2: An Improved jQuery.when” on page 63](#), we met `when2` and saw how it could be used to time out a single promise. One of the challenges given in that example was figuring out why that approach will not work if you need to time out multiple simultaneous deferreds.

The problem is that setting `resolveOnFirstSuccess` to true in calling `when2` will prevent us from being notified properly when all the promises have been resolved. The timeout part works fine. The problem is that any single promise that completes before the timeout will trigger the resolve of the master deferred from `when2`. There's no way to get all the results!

A sneaky first thought for fixing this might be to instead use a modified `wait` function that *rejects* its deferred if the timeout happens:

```

function waitThenReject(timeout){
  var d = $.Deferred();
  setTimeout(d.reject, timeout);
  return d.promise();
}

```

Then we could use it as follows, assuming we already had two promises returned from other functions:

```

/* THE FOLLOWING DOESN'T WORK! */

$.when2([waitThenReject(200), promise1, promise2])
  .done(function(timeoutResult, result1, result2){
    // Both promises are resolved.
  })
  .fail(function(index, error){
    if (index === 0){
      // Timeout!
    }
    else {
      // The promise given by index was rejected with the given error.
    }
  });

```

But, cute as it is, this approach has a fatal flaw. Can you spot it?

The problem is that the promise returned by `waitThenReject` is never resolved—it can only be rejected. The `done` callback above will never be called. In fact, the promise returned by `when2` will *always* be rejected, even if `promise1` and `promise2` are both resolved before the timeout. If they both resolve early, the deferred created by `when2` doesn't get resolved as it is waiting for *three* deferreds to resolve.

One solution to this would be to add an integer `resolveAfter` to the options passed to `when2` that would cause it to resolve the master deferred once that many underlying promises were resolved. Part of that is easy (the resolving), but part of it would be slightly messy—what values would we resolve the master deferred with for those promises that had not yet resolved?

Instead, let's do something more challenging to help you learn more about working with deferreds. First, here's a timeout function that returns a promise and a trigger function:

```

function deactivatingWait(timeout, triggerCount){
  var deferred = $.Deferred(),
      count = 0,
      timeoutId = setTimeout(deferred.reject, timeout),
      trigger = function(){
        if (++count === triggerCount){
          clearTimeout(timeoutId);
          deferred.resolve();
        }
      };
}

```

```

    }
    return [deferred.promise(), trigger];
}

```

As with our `waitThenReject` function, the returned promise will be rejected after the timeout has elapsed.

However, this will only happen if the trigger function has not already been called at least `triggerCount` times. If the trigger is called `triggerCount` times, the returned promise is resolved.

As a result, we obtain a promise that will be rejected after a timeout, but which can also be made to resolve under the right conditions. All we need to do is call `deactivatingWait`, arrange for our preexisting promises to call the trigger function when they're resolved, and then pass the promise from `deactivatingWait` to `when2` along with our original promises:

```

var result = deactivatingWait(200, 2),
    timeoutPromise = result[0],
    trigger = result[1];

promise1.done(trigger);
promise2.done(trigger);

$.when2([timeoutPromise, promise1, promise2])
  .done(function(timeoutResult, result1, result2){
    // Both promises resolved before the time out.
  })
  .fail(function(index, error){
    if (index === 0){
      // Timeout!
    }
    else {
      // The promise given by index was rejected with the given error.
    }
  });

```

The above is slightly clunky, but we're almost there. Now that we know what to do, it's not hard to create a utility function to hide the setup work and to massage the `when2` callback results:

```

function when2WithTimeout(timeout, promises, options){

  var result = deactivatingWait(timeout, promises.length),
      timeoutPromise = result[0],
      trigger = result[1],
      i;

  for (i = 0; i < promises.length; i++){
    promises[i].done(trigger);
  }
}

```

```

    return $.when2([timeoutPromise].concat(promises), options).then(
      function(){
        // Drop the 'undefined' result of the deactivatingWait promise.
        return Array.prototype.slice.call(arguments, 1);
      },
      function(index, error){
        // Adjust the index to match promises.
        return [index - 1, error];
      },
      function(index, value){
        // Adjust the index to match promises.
        return [index - 1, value];
      }
    );
  }
}

```

This function is used as follows:

```

when2WithTimeout(200, [promise1, promise2])
  .done(function(results){
    // Both promises are resolved. Results are in results[0] and results[1].
  })
  .fail(function(result){
    var index = result[0],
        error = result[1];
    if (index === -1){
      // Timeout!
    }
    else {
      // The promise given by index was rejected with the given error.
    }
  });

```

The timeout is detected in the fail callback when the index is -1.

Challenges

1. What would happen if we removed the `clearTimeout` call in the `deactivatingWait` function above?
2. A minor wrinkle in `when2WithTimeout` is that it should check options to make sure `rejectOnFirstError` is true (or missing, since that's the default). What would happen if `rejectOnFirstError` were not true?

Controlling Your Own Destiny

Through this book we've drilled the *create deferreds*, *return promises* message into you. That's a safety-first convention that makes good sense: code that receives a promise has no way of accidentally firing it.

But what if you want to *deliberately* fire a promise you receive?

While this might sound like it goes completely against the intention of deferreds, in practice there are legitimate reasons you might want to fire a promise you received from someone else.

For example, you might be building a system that needs to time out the requests it is making, causing them to immediately resolve with a specific default value. Or you might want an outstanding deferred to be rejected at a specific moment (e.g., system shutdown) with a specific given value. Or maybe you've received a promise for a long-running task that doesn't send any progress information and you want to provide progress information (e.g., elapsed time) yourself by calling `notify`. You can't do any of these things if all you have is a promise. The code that returned you the promise has total control. If there's a bug in that code that causes it not to fire, or something else goes wrong, you're out of luck and your whole program may hang as a result.

So there are plenty of reasons why you might want to somehow resolve, reject, or notify a promise you were given. Not letting you do so just so you don't accidentally shoot yourself in the foot is a pretty thin reason to take away control. Wouldn't it be more mature and trusting if code that returned a read-only promise instead returned a deferred? The deferred receiver could convert it into a read-only promise if it knew it would never need to fire the deferred and wanted to disable the `resolve`, `reject`, etc., methods for safety?

Given that we're unlikely to change the world and that there is already a ton of code out there that returns promises, what can we do? How can we wrest back control and escape the read-only world where we're treated like children who need to be protected from themselves? Take a moment to think about it.

It turns out there's an easy fix that again illustrates the elegance of deferreds. You can write a function like the following:

```
function deferredFromPromise(promise){
  var deferred = $.Deferred();

  promise
    .done(deferred.resolve)
    .fail(deferred.reject)
    .progress(deferred.notify);

  return deferred;
}
```

How cool is that?

The `deferredFromPromise` function takes a promise and returns a new deferred. All it does is hook up the output (`done`, `fail`, `progress`) of the original promise so that it will trigger the corresponding method (`resolve`, `reject`, `notify`) on the deferred. Because there's no problem if a jQuery deferred has `resolve` or `reject` called on it multiple times (the later calls have no effect), only the first code to fire the deferred returned by `deferredFromPromise` will determine its fate. That could be the code that made the deferred from which the original promise was obtained (the normal situation), but it could equally be *your code* that receives the deferred from `deferredFromPromise`.

This is another example of why deferreds are so elegant. We had what looked like a fundamental limitation but found an easy way around it by *making another deferred* and wiring things up the way we wanted them. Other examples of this technique are the promise then method, `$.when` (if you pass it more than one argument), `when2` (see “[when2: An Improved jQuery.when](#)” on page 63), and our example in “[Timing Out Promises](#)” on page 69. Deferreds make great building blocks!

Challenges

1. Rewrite `deferredFromPromise` to be a function that executes just one statement.
2. If you were writing an API that returned promises and you wanted to give your caller the ability to fire them, you could simply wrap your returned promise in a call to `deferredFromPromise`. What could you do that would be even easier?
3. What will happen if promise has already been resolved or rejected when it is passed to `deferredFromPromise`?

Deactivating a Promise

To complement “[Controlling Your Own Destiny](#)” on page 73, there's another way in which you might want to give yourself more control over a promise.

What happens if for some reason you decide you're no longer interested in the result of a promise and you don't want to receive a value from it at all? For example, you might be shutting down a system and need to ignore some outstanding promises that you would normally want to react to. A subsystem that could handle a resolved promise result might have been shut down or a data structure holding results might no longer exist. Whatever the case, you want to deactivate an existing promise, so that it never calls the callbacks you attached to it earlier.

Once again we can solve the problem using a second deferred. Here's a deactivatable Promise function that takes one promise and returns another that has a deactivate method on it:

```
function deactivatablePromise(promise){
  var deferred = $.Deferred(),
      newPromise = deferred.promise(),
      deactivated = false, // ❶
      unlessDeactivated = function(func){ // ❷
        return function(value){
          if (deactivated === false){
            return func(value);
          }
        };
      };

  promise // ❸
    .done(unlessDeactivated(deferred.resolve))
    .fail(unlessDeactivated(deferred.reject))
    .progress(unlessDeactivated(deferred.notify));

  newPromise.deactivate = function(){ // ❹
    deactivated = true;
    return newPromise;
  };

  return newPromise; // ❺
}
```

Suppose you call a function that returns a promise and you want to be able to deactivate the promise if it becomes necessary.

Instead of adding callbacks to the promise, first pass it to `deactivatablePromise` to obtain a promise that you can deactivate. Then add your callbacks to that new promise. If you later call the `deactivate` method on it, the new promise will not call any of its callbacks when it fires (unless it had already fired, of course).

Here's how it works:

- ❶ We use a `deactivated` variable that keeps track of whether `deactivate` has been called and that we'll use to conditionally trigger the new promise.
- ❷ The `unlessDeactivated` function takes a function argument (`func`) and returns a function that will call `func` with an argument, but only if `deactivate` has not been called.
- ❸ Arrange for the existing promise to conditionally call the new deferred's `resolve`, `reject`, and `notify` functions if it fires. If the new promise has been deactivated (see next point), though, the event will not be passed on to the new promise.

- ❹ We add a `deactivate` method to the new promise that just sets the `deactivated` variable to be true.
- ❺ Return the new promise.

Challenges

1. Why does the `newPromise.deactivate` function return the new promise?
2. `unlessDeactivated` returns a function that takes one parameter (`value`). Change it to be compatible with jQuery deferreds: it should accept any number of parameters and pass these on in the call to `func`.
3. It's not 100% safe to add our own `deactivate` method to the promise returned by `deactivatablePromise`. If a future version of jQuery adds an identically named function to promises, we'll clobber it. How might you change `deactivatablePromise` to eliminate this risk?
4. Write a function that takes a promise and returns an object with `resolve`, `resolveWith`, `reject`, `rejectWith`, `notify`, and `notifyWith` methods (as in “[Controlling Your Own Destiny](#)” on page 73) as well as a `deactivate` method, as in this example. That combination gives you total freedom: you can (essentially) fire the promise yourself or you can choose to deactivate it.

More Time in the Mental Gymnasium

The recipes in the previous chapter are useful in a practical sense, yet we also wrote them to help you exercise the mental muscles needed for thinking about deferreds. Although conceptually simple, deferreds and promises can be mind-bending and yet a pleasure to ponder.

It may take some time to digest everything, but deferreds are worth exploring further (both more deeply and broadly), and that's the purpose of this chapter.

Do You Really Understand jQuery Deferreds?

To warm up, let's see if you really understand jQuery deferreds. If you do, you should be able to answer the following questions with little hesitation:

- Explain the differences between `done`, `then`, and `$.when`.
- Given a `promise` variable, what's the difference between these two code fragments?

```
promise.then(doneFunc, errorFunc)
```

```
promise.done(doneFunc).fail(errorFunc)
```

- What happens if you return a value from a `done` callback? What should you probably be doing instead?
- What happens if you call `then` on a promise, but don't provide any arguments?
- Take another look at this code fragment from **"Familiar Promises" on page 4**:

```
$.when(  
  $.ajax('http://google.com').then(function(){  
    return $('#label_1').animate({ opacity: 0.25 }, 100);  
  }),  
  $.ajax('http://yahoo.com').then(function(){  
    return $('#label_2').animate({ opacity: 0.75 }, 200);  
  })  
)
```

```

).then(
  function(){
    // Both URLs have been fetched and both animations have completed.
  }
);

```

In order of difficulty:

- `.promise()` is not called on the result of the calls to `animate`. Why is that not needed?
- What would be the effect of removing the two `return` keywords (i.e., call `animate` as above, just don't return its result)?
- How can you arrange to get the results of the `$.ajax` calls passed as arguments to the final function?

If you're unsure of the answers to these questions, rereading [Chapter 2](#) will probably help, as will writing small pieces of code to have a closer look. Make sure you understand all the points listed in [“Deferred Dynamics” on page 18](#).

Promises/A+

Feeling warmed up? Let's stretch our mental muscles a bit further by learning about a standard for promises.

We have mentioned the [Promises/A+ proposal](#) at several points in the book. It's a library-agnostic clarification of the expected behavior of promises. There is also the original [Promises/A](#) (upon which jQuery promises is based) and a newer [Promises/B proposal](#). Promises/A+ is the most popular current proposal. A+ is a clarification, extension, and simplification of the original Promises/A.

It is important that you know about Promises/A+, because most of the popular promise libraries conform to the A+ proposal. Its main (although not exclusive) focus is the means of interacting with promises via the `then` method. It's a short, well-written document released in the public domain. It takes only minutes to read. We've included a copy in [Appendix B](#).

To summarize, Promises/A+ states that the `then` method should return a new promise. In so doing, it facilitates the chaining of callbacks via further calls via `then` on subsequently returned promises. It makes explicit the expected behavior of both the `onFulfilled` and `onRejected` handlers and how a promise may transition between pending, fulfilled, and rejected states.

Rather than repeat an already well-written and compact document, we feel it necessary to explain the most important and subtle way it differs from the apparently similar jQuery deferred API.

It boils down to error handling.

In the Promises/A+ world, an exception raised in a callback passed to `then` is handled automatically. The call to `then` will return a promise in a rejected state, with the exception set as the `reason` attribute of the new promise.

However, jQuery promises don't do this. Their `then` doesn't return a rejected promise. Instead, exceptions are not caught. An exception in a function passed to `then` will break any neatly composed chain of `then` handlers. An exception in an `always`, `done`, or `fail` callback will terminate that callback but also prevent the invocation of the rest of the callbacks added to the promise. In all these cases, the exception will propagate up the call stack. If you're running a node.js server, an uncaught exception will cause your server to exit, which is clearly a serious problem.

Therefore, when using jQuery deferreds you'll need to be more careful about catching regular exceptions in your callback functions and finding ways to fail when the original call stack context (in which you arranged for the callbacks to be invoked) has long gone. Promises/A+ lets failures flow back via rejected promises.

This difference and its impact is elegantly discussed in Dominic Denicola's article "[You're Missing the Point of Promises](#)", in which he introduces a **test suite** for Promises/A+ compliance. The slides of Dominic's "[Promises, Promises](#)" presentation are also worth reading, particularly the description of the correspondence between asynchronous Promises/A+ code and regular synchronous code.

There are a number of other differences between jQuery deferreds and Promises/A+. For example, the jQuery version of `then` allows you to pass a progress function, but Promises/A+ does not require this (though Promises/A did). The Promises/A+ specification makes no mention of methods such as `done`, `fail`, etc., relying only on `then`. You will discover other differences—including runtime ones such as speed and memory usage—if you compare the implementations yourself or spend some time online reading comparison tests and benchmarks.

There are *many* packages for JavaScript (and some in other languages too) that conform to the Promises/A+ standard. The Promises/A+ website contains [an up-to-date list](#).

Promises Are First-Class Objects for Function Calls

In JavaScript, we describe functions as first-class objects. It's why we're able to do all sorts of interesting, empowering, and downright fun things with functions: return a function from a function, pass a function as an argument into yet another function, and create anonymous functions on the fly. It's hard to imagine JavaScript without such a powerful core feature.

With that in mind, try this exercise in a change of perspective: deferreds and promises are first-class objects for *function calls*.

If you think about it, they give us a way to take a function call (or the handling of an event) and pass it around: we can return a promise from a function, pass a promise as an argument into a function, store it in a data structure, and, obviously, arrange how to process the result (or error) of the function call independently of the function call itself.

To be a bit more abstract: *a promise is a time-independent reification of a function call*.

A promise is time-independent because, as we saw in “Memoization” on page 49, you don’t have to worry whether the underlying function call has already completed, is currently in progress, or has yet to run. There may even be no way to find out. The point is that *you don’t have to care and you shouldn’t care*.

Like many others, we described deferreds and promises in [Chapter 1](#) as representing a result that may not be available yet. Describing promises as a reification of a function call takes the time factor completely out of the picture. If you’ve managed to switch your perspective, this way of looking at deferreds is a lot simpler to explain and understand.

Asynchronous Data Structures

Although deferreds are useful for handling familiar tasks such as network calls, user interface events, and talking to databases and other storage, nothing about deferreds is specific to those things. Because deferreds are not tied to any particular usage, you can use them to build asynchronous implementations of other things that might seem surprising.

For example, consider asynchronous data structures. “A Deferred Queue” on page 61 gives a good example of this line of thinking. Let’s revisit the deferred queue code, slightly simplified:

```
var waiting = [], queue = [];

function get(){
  var deferred = $.Deferred();
  if (queue.length){
    deferred.resolve(queue.shift());
  }
  else {
    waiting.push(deferred);
  }
  return deferred.promise();
}

function put(item){
  if (waiting.length){
    waiting.shift().resolve(item);
  }
}
```



```
    else {  
        queue.push(item);  
    }  
}
```

As we suggested in that example, think about code for a regular synchronous queue. What happens if you call `get` on a regular queue that's empty? Almost certainly one of two things: you'll get some kind of `QueueEmpty` error, or else your code will block until some other code puts something into the queue. That is, you either get a synchronous error or you get a synchronous nonempty response.

In contrast, look at the `get` function above. A new deferred is always made. If the queue is nonempty, the deferred is resolved and its promise is returned. If the queue is empty, the deferred is added to `waiting` and its promise is returned. Code calling `get` on an empty queue doesn't get an error and it doesn't block; it always receives a result immediately. How can you get a result from an empty queue? Easy! The result is a promise.

In either case, the caller just attaches callbacks to the returned promise, and goes on its merry way. It doesn't need to know if the promise has already fired or when it will fire. It just needs to know how to process a queue item.

In effect, we've built an asynchronous data structure. We find that concept very attractive, and thinking in those terms helped us to see part of the reason why deferreds are so great. Deferreds are not specifically asynchronous or synchronous: they're so simple they don't even care, and as a consumer of promises, your code doesn't have to either. That's all remarkably elegant.

Bonus challenge: imagine you have a producer and a consumer wanting to exchange a **dictionary data structure**. The producer will produce values in a random key order, while the consumer requests keys in some other random order. Implement an asynchronous dictionary using deferreds in the style of the deferred queue above.

Advantages of Deferreds

It is typical to first gain an appreciation of deferreds (in general, not just the jQuery flavor) by using them in the context of a specific problem. But deferreds are useful and powerful in multiple ways. Your understanding will be stronger if you consider them all at once. Here's a summary.

Deferreds provide a formal first-class place to hold a future result. You can pass a deferred to other pieces of code. This allows multiple independent pieces of code to act on its result. The first-class object gives your program a data structure that captures the fact that a function has been called. That data structure can be used from the moment the function has been called until it has returned. As we saw in [“Memoization” on page 49](#), this can be very useful. Because the deferred is time-independent, you also don't

need to worry about events firing before you’ve added handlers for them—you can add event handlers (callbacks) after the fact.

Without deferreds, you get to attach just one callback handler to events. If other code needs to also handle the result of the event, calls to those other functions need to be placed in the original callback. This leads to a mess: callback handlers that do not serve a single purpose and that are harder to understand and test.

Deferreds make great building blocks. We’ve repeatedly seen how simple it is to solve problems by creating an additional deferred and hooking up things just as we need them. It is a testament to the elegance and power of deferreds that this can be done so cleanly and easily. Each time you do it, you just end up with another deferred, which allows for additional processing layers to be added in an identical way, if needed.

Deferreds make it simple(r) to work with multiple asynchronous calls. In “[Food for Thought](#)” on page 1 and “[when](#)” on page 13, we looked at how awkward it can be to work with more than one asynchronous function call. The examples we considered were trivial. When building non-toy systems, things get much more complicated. Deferreds can’t make a complicated situation less complex, but they will allow you to write code to handle it in a cleaner way. Code that doesn’t use deferreds does not have an object to pass around that represents the handling of an event (or the future result of a function call). This forces us to set up complex event handling using the lexical structure of our code—callbacks in callbacks in callbacks, etc. The logical structure of one’s code can become extremely awkward, as can error handling. Complex programs using deferreds can also be very difficult to understand and arguably harder to follow logically, but there is more scope for cleaner separation of code and nicer error handling (at least with Promises/A+).

Deferreds are very useful for setting up processing of request responses in a free-form messaging environment (where both sides can send messages at any time). We saw examples of this in “[Communicating with a Web Worker](#)” on page 37 and “[Using Web Sockets](#)” on page 42. An important advantage of this approach is that code that makes an asynchronous call can arrange to process the response without losing context. When you make the call, your code has access to its current scope, local variables, relevant UI components, etc. In free-form messaging, when the reply arrives, you often need to reconstruct the necessary parts of the original context in order to process the result. With deferreds, you can set this up on the spot by adding handlers to the deferred, which is extremely convenient in practice.

Difficulties with Deferreds

There are a few ways in which deferreds can be problematic.

Occasionally you will build a complex structure of dependent deferreds and one of them will fail to fire. For example, a network request is somehow lost or never returns. This

jams up the whole system: a deferred never fires and all downstream deferreds fail to resolve. This kind of problem has to be dealt with by programmers in any case; it's not specific to deferreds, but there can be a feeling of a certain brittle dependency. We've explored several ways in which these problems can be addressed: timing out promises (in [“when2: An Improved jQuery.when” on page 63](#) and [“Timing Out Promises” on page 69](#)), firing promises ourselves (in [“Controlling Your Own Destiny” on page 73](#)), and deactivating promises entirely (in [“Deactivating a Promise” on page 74](#)).

A second issue, which is more serious and which may preclude the use of deferreds at all, is that deferreds keep the result of their computation around. If it happens that you only need a result once, introducing deferreds will add to memory load and may make garbage collection more difficult for the system.

Deferreds are often viewed as resulting in code that is difficult to read and understand. This is subjective, of course. It is also the case that some programming situations are inherently complicated, and that complexity cannot really be mitigated. Nevertheless, making sense of deferred code can sometimes be more difficult than it is with more traditional JavaScript callback nesting. This is especially the case with very simple event handling. Once asynchronous code logic gets more than a couple of levels deep, we've found deferred code to be cleaner due to its separation and having first-class objects to pass around and store in data structures. There's more to deferreds, though, than potential gains (or losses) in code organization and readability, as just discussed above. In our experience, any loss in code readability is more than made up for by gains in other ways.

There is sometimes the feeling of a lock-in effect when using deferreds. Once you start calling and writing functions that use deferreds, you'll invariably end up doing more of it. That's also the case with nested callback handling; it's just a different style of getting things done. Fortunately, JavaScript is inherently event-based, so it is usually trivial to write deferred-based wrappers for functions you want to call that do not follow this style. It is also usually very easy to mix deferred-using code with regular code.

Finally, if you're building a team, use of deferreds may have a human resources impact: you'll likely end up looking for programmers who are already familiar with deferreds. Programmers who are *unfamiliar* with deferreds are definitely still in the majority. We hope to help change that!

Further Reading

James Coglan has written three excellent articles on promises: [“Promises are the monad of asynchronous programming”](#), [“Callbacks are imperative, promises are functional: Node's biggest missed opportunity”](#), and [“Callbacks, promises, and simplicity”](#). The articles may not be easy to understand on a first reading. You might find it easiest to start with the second. They are well worth rereading and thinking about until you *do*

get it. Don't give up—it's important! For balance, you should also read [Drew Crawford's](#) response, "[Broken Promises](#)", which describes issues he ran into using promises in the highly memory-constrained iOS environment, and, more generally, why trying to use promises to solve every asynchronous flow of control problem can lead to difficulties.

Many people have written about jQuery deferreds online, and some of the articles are excellent. There was a flurry of writing when deferreds were introduced to jQuery in version 1.5 on January 31, 2011. As mentioned in "[Changes in the jQuery Deferred API](#)" on [page 19](#), subsequent changes in the deferred API mean you need to be a little careful when reading. For example, most recent online articles still discuss the `pipe` function, which was deprecated in version 1.8 (August 9, 2012). But don't worry too much: even if articles are slightly out of date with respect to the API, you can still learn a lot from them and may benefit from other people's perspectives and explanations. Here's a small selection, focused mainly on jQuery deferreds, that you may find useful:

- "[Creating Responsive Applications Using jQuery Deferred and Promises](#)"
- "[Redemption from Callback Hell](#)"
- "[jQuery.Deferred is the most important client-side tool you have](#)"
- "[Using Deferreds in jQuery 1.5](#)"
- "[Asynchronous Programming in JavaScript with 'Promises'](#)"

You might also be interested to read how [Twitter transitioned their architecture](#) to use "services" that return futures (i.e., deferreds) via RPC using [Finagle](#).

Deferreds have been around since 1976, under many names and guises, and in many programming languages. For some history, see the Wikipedia article on "[Futures and promises](#)".

Hints for Selected Challenges

Here you'll find solutions or hints for almost all the challenges in the book. Where an answer requires a significant amount of code, we make some suggestions but have left the coding to you.

A Replacement for the `setTimeout` Function

1. This is pretty easy:

```
function wait(timeout){
  return $.Deferred(function(deferred){
    setTimeout(deferred.resolve, timeout);
  }).promise();
}
```

2. You'd write something like this:

```
var promise = wait(500);

promise.done(function(){
  console.log('Timeout fired!');
});

one(promise);
two(promise);
```

Of course the one and two functions have to be able to accept a promise.

We described this challenge as “subtly different” because it's meant to demonstrate that our replacement `setTimeout` function is more than just a syntactic nicety. It gives you something you didn't have before: a value that you can pass around to others. Although this is a trivial example, the principle is important.

3. We just need to store the timeout identifier and add a function to the returning promise to clear the timeout and reject the deferred:

```
function wait(timeout){
  var deferred = $.Deferred(),
      promise = deferred.promise(),
      timeoutId = setTimeout(deferred.resolve, timeout);

  promise.cancel = function(){
    clearTimeout(timeoutId);
    deferred.reject.apply(deferred, arguments);
  };

  return promise;
}
```

Note that this isn't the best coding style. We probably shouldn't be adding attributes to the promise in this way. How would you fix this shortcoming? While you're at it, add a `cancelWith` function to call `rejectWith` on the deferred.

Messaging in Chrome Extensions

1. That depends. The message to the content script will never receive a reply, so the promise will never be resolved. That's fine if the caller isn't expecting a response and doesn't need to know that the message was received. If there's good coordination between the people writing the background code and the content script code, it may be perfectly fine not to send a response. The safest route is probably to always send back a null response to indicate that the message was received, though this adds communication overhead within Chrome.
2. The results from the tabs are collected by `$.when` and passed as arguments to the done callback function. The simple code we showed in the example just happens to ignore its arguments.

Accessing Chrome Local Storage

1. This is pretty straightforward:

```
function get(keys){
  var deferred = $.Deferred();

  chrome.storage.local.get(keys, function(result){
    if (chrome.runtime.lasterror){
      deferred.reject(chrome.runtime.lasterror.message);
    }
    else {
      deferred.resolve(result);
    }
  });
}
```

```

        return deferred.promise();
    }

    function getBytesInUse(keys){
        var deferred = $.Deferred();

        chrome.storage.local.getBytesInUse(keys, function(bytesInUse){
            if (chrome.runtime.lasterror){
                deferred.reject(chrome.runtime.lasterror.message);
            }
            else {
                deferred.resolve(bytesInUse);
            }
        });

        return deferred.promise();
    }

    function remove(keys){
        var deferred = $.Deferred();

        chrome.storage.local.remove(keys, function(){
            if (chrome.runtime.lasterror){
                deferred.reject(chrome.runtime.lasterror.message);
            }
            else {
                deferred.resolve();
            }
        });

        return deferred.promise();
    }

    function clear(){
        var deferred = $.Deferred();

        chrome.storage.local.clear(function(){
            if (chrome.runtime.lasterror){
                deferred.reject(chrome.runtime.lasterror.message);
            }
            else {
                deferred.resolve();
            }
        });

        return deferred.promise();
    }

```

2. The **node.js Redis** package provides an API call for the Redis **get** command. The node.js Redis API signature is:

```
get('hash-key', function(err, result) {
    // Check for error, use result, etc.
});
```

The callback function is called with two arguments, an error and a result, as is common in node.js APIs. The caller should check to see if an error occurred and if not, it can use the result.

Writing a promise-returning replacement is quite straightforward:

```
var redis = require('redis'),
    client = redis.createClient(),
    $ = require('jquery-deferred');

function redisGet(hashkey){
    var deferred = $.Deferred();

    client.get(hashkey, function(err, result){
        if (err){
            deferred.reject(err);
        }
        else {
            deferred.resolve(result);
        }
    });

    return deferred.promise();
}
```

There are of course many other Redis API calls you could make promise-returning versions of. Not surprisingly, you can also find a full **promise-based API**, though note that it is based on **Promises/A+**, not on jQuery deferreds.

Running Promise-Returning Functions One by One

1. In the loop in synchronously, as well as shifting out the task function, also shift a context. Pass the context to `apply` as its first argument (instead of passing `null`). When you call synchronously, you'll obviously need to pass in a context for each function.
2. Any arguments to the function passed to `then` are ignored because they will contain the return result of the previously run function (or `undefined` when the first function is run). The values are ignored because we are not passing along a modified result, we're just running the functions (whose arguments are passed in the tasks list and given to `apply`).
3. To allow for calling functions that do not return promises, we need to check the return result from `apply`. If it has a method called `promise` that is a function, we

can call it and return the result. Otherwise, we just return the value that comes back from `apply`. Note that this is also how jQuery guesses that a value is a deferred. It's not foolproof!

This is pretty much what `$.when` does when you pass it a single argument, so you could just make a simple change and use `$.when`:

```
function synchronously(tasks){
  var i, task, func,
      promise = $.Deferred().resolve().promise(),
      makeRunner = function(func, args){
        return function(){
          return $.when(func.apply(null, args));
        };
      };
  for (i = 0; i < tasks.length; i++){
    task = tasks[i];
    func = task.shift();
    promise = promise.then(makeRunner(func, task));
  }
  return promise;
}
```

4. If `done` is used instead of `then`, the promise will fire all its attached callbacks as soon as the first animation is complete. That is, it will not wait until the second animation is finished. The second animation is a `done` callback on the promise, just like any other. This is a common source of error, and can be hard to debug. A solid understanding of deferred dynamics (see “Deferred Dynamics” on page 18) will make you less likely to write code like this in the first place.
5. The `$.when` version will start both animations at the same time instead of running them one after the other.

It is not necessary to call `promise()` on the result of `animate` because `$.when` will do that for you. Note that `animate` doesn't return a promise directly; it's necessary to call `promise()` on its result to get one.

A Promise Pool with an emptyPromise Method

1. Create a deferred, initialize it, and return its promise in one statement:

```
function emptyPromise(){
  return $.Deferred(function(deferred){
    waitingForEmpty.push(deferred);
  }).promise();
}
```

2. The promise returned by `emptyPromise` will not be resolved until after at least one promise is added to the pool and the pool subsequently empties.

3. Change the emptyPromise function as follows:

```
function emptyPromise(checkImmediately){
  var deferred = $.Deferred();
  if (checkImmediately && inProgress.length === 0){
    return deferred.resolve().promise();
  }
  else {
    waitingForEmpty.push(deferred);
    return deferred.promise();
  }
}
```

4. The shutdown handler could look like this:

```
app.get('/shutdown', function(req, res){
  if (shuttingDown === false){
    shuttingDown = true;
    pool.emptyPromise().done(function(){
      pool.add(flushAvatarCache());
      pool.add(sendShutdownEmail());
      pool.add(flushLogs());
      pool.emptyPromise().done(function(){
        res.send(200);
        process.exit(0);
      });
    });
  }
  else {
    res.send(200);
  }
});
```

5. This one's up to you!
6. You could maintain an integer request ID, incremented on each call to add. Use a closure to capture its current value when setting up the always callback. The ID would be an attribute of inProgress (now a JavaScript object). So when a promise is resolved or rejected, the promise can immediately be found in inProgress. You could maintain a count of the pool size to know when to resolve waiting deferreds, or else use the jQuery convenience function `isEmptyObject`.
7. Keep a list of responses that are awaiting an HTTP 200 status reply (shutdownResponses in the code below). When the first shutdown request is received, arrange to later respond to all shutdown requests (including those which arrive during the shutdown) in the done callback that runs after the pool empties for the second time. You might try something like this:

```
var express = require('express'),
    app = express(),
    pool = createPromisePool(),
    shuttingDown = false,
```

```

    shutdownResponses = [];

    app.get('/', function(req, res){
        if (shuttingDown){
            res.redirect('http://example.com');
        }
        else {
            pool.add(handleRequest(req, res));
        }
    });

    app.get('/shutdown', function(req, res){
        shutdownResponses.push(res);

        if (shuttingDown === false){
            shuttingDown = true;
            pool.emptyPromise().done(function(){
                pool.add(flushAvatarCache());
                pool.add(sendShutdownEmail());
                pool.add(flushLogs());
                pool.emptyPromise().done(function(){
                    for (var i = 0; i < shutdownResponses.length; i++){
                        shutdownResponses[i].send(200);
                    }
                    process.exit(0);
                });
            });
        }
    });

    app.listen(9999);

```

Note that we again use `emptyPromise` twice.

Bonus challenge! If the `shutdownResponses.push(res);` line were moved after the `if` block that follows it, we might be introducing a subtle bug. Can you see what it is? Hint: What happens if, on the first shutdown call, the request pool is already empty and the three administrative cleanup functions complete instantaneously?

Displaying Google Maps

1. There are tons of JavaScript LRU cache packages around. Use one that provides a key/value store API. The keys in our case can be a latitude/longitude string and each value will be a data URI. The LRU cache will replace `lastKey` and `lastDataURI`.
2. When a data URI has been retrieved from Google, `getMap` calls `localStorageSet`, but discards its return value.

We did this for simplicity—and because there’s not much that could be done at this point, other than logging or perhaps retrying a failed call. If the call fails and the last value isn’t cached as a result, it’s not fatal—the application continues to work, albeit with less effective caching. In production code you’d probably want to do *something*, but showing an error message to a user is probably out of the question and logging will likely go unnoticed. Maybe the error could be silently reported via a server call.

3. Sorry, you’re on your own!
4. Use the example in “[Short-Term Memoization of In-Progress Function Calls](#)” on [page 52](#) to keep track of in-flight Google API requests.

Communicating with a Web Worker

1. Log the message in an else branch you add to the if (requests.hasOwnProperty(id)) in handleResponse.
2. If you’re not sure where to keep the timestamp for each request, the answer to the next challenge will help. The reason the request times build up is that the web worker is also run in a single thread and can only do one thing at a time.
3. Here’s how we’d do it:

```
function createWebWorker(sourceFile){

  var queue = [], // IDs of requests yet to be sent.
      requests = {}, // Key is request ID, value is request spec.
      requestId = 0, // Incremented on each message sent.
      worker = new Worker(sourceFile),
      workerState = 'idle',

      processQueue = function(){
        var id, request;
        if (workerState === 'idle' && queue.length > 0){
          workerState = 'busy';
          id = queue.shift();
          request = requests[id];
          worker.postMessage({
            method: request.method,
            payload: request.payload,
            requestId: id
          });
        }
      },

      enqueueRequest = function(method, payload){
        var deferred = $.Deferred(),
            id = requestId++;
```

```

        requests[id] = {
            deferred: deferred,
            method: method,
            payload: payload,
        };
        queue.push(id);
        processQueue();
        return deferred.promise();
    },

    handleResponse = function(response){
        var deferred, id;
        if (response.hasOwnProperty('requestId')){
            workerState = 'idle';
            id = response.requestId;
            if (requests.hasOwnProperty(id)){
                deferred = requests[id];
                delete requests[id];
                if (response.hasOwnProperty('result')){
                    deferred.resolve(response.result);
                }
                else {
                    deferred.reject(response.error);
                }
            }
            processQueue();
        }
        else {
            // An unsolicited message from the worker.
            if (response.type === 'log'){
                console.log('Worker says:', response.message);
            }
            else {
                console.log('Unknown message from worker:', response);
            }
        }
    }
};

worker.addEventListener('message', function(event){
    handleResponse(event.data);
});

return {
    run: enqueueRequest
};
}

```

We've added a queue of request IDs that have yet to be sent to the worker. To make the code a bit more general, we just return an object with a run function. run takes a method name and a payload and passes these to enqueueRequest, which puts a request specification into requests and adds the request ID to queue. It then calls

processQueue, which will send a request to the worker if it's not busy. Each time we handle a response, processQueue is called again.

4. Based on the code above: in enqueueRequest, store the time the request is received into the requests object, along with the rest of the request specification. In processQueue, add another timestamp to the request specification to record when the queued request is sent to the worker. In handleResponse, log both the time spent in the queue and the time spent processing.
5. Also based on the above code: keep an array of workers, each with a state. In processQueue, look for an idle worker, mark it as busy, and send it the request. In handleResponse, you'll need to figure out which worker sent the response so you can mark that worker as now being idle. That means you'll need to save the worker's index in the request specification in enqueueRequest.

Using Web Sockets

1. Instead of processing the return result of func(request.payload) using done and fail callbacks, use \$.when, which knows how to handle a nonpromise result, returning it via a new promise.
2. If the server didn't send a request ID in a response, the value of deferred in the handleResponse function in the web sockets client will be undefined, which will cause an exception to be raised. The deferred corresponding to that request (assuming the server has not spontaneously sent a response without receiving a request) will never fire, so the request will hang indefinitely unless timed out by some other mechanism. The same thing will happen—though for a slightly different reason—if a request ID that has already been used is sent.
3. You're on your own! Obviously, the server needs to put its own request IDs into the messages it sends, and the client will need to send them back. You'll need to make sure that both sides can distinguish between responses to their own outstanding requests and incoming new requests from the other side.
4. The problem is the use of \$.when. There is no server-side multiply function, so the corresponding makeRequest promise will be rejected. That causes the immediate rejection of the promise returned by \$.when. You could fix this by using when2 (see [“when2: An Improved jQuery.when” on page 63](#)) and setting the rejectOnFirstError option to false.
5. The jQuery source code line that causes \$.when to reject immediately is in the for loop near the bottom of the when function in [deferred.js](#). It's this line: `.fail(deferred.reject)`.

If the call signature of `$.when` had accepted a *list* of promises instead of treating all its arguments as promises, it would have been possible to alter its behavior with a flag. See “[when2: An Improved jQuery.when](#)” on page 63 for an approach to changing this behavior.

Automatically Retrying Failing Deferred Calls

1. This is too easy to warrant even a hint!
2. The problem occurs if the passed function returns a promise that is rejected with no arguments. In that case, the value of `value` in the `error` function will be `undefined`. If a subsequent call to the passed function returns a promise that is rejected with a non-`undefined` value, that failure result will be stored in `rejectValue`. In this case, the first failure result (`undefined`) is overwritten and will not be returned. You can fix this small problem by introducing another variable, to remember whether a failure has occurred. Instead of testing the value of `rejectValue` in `error`, test the new variable—and only if no previous failure has occurred, store the passed error value in `rejectValue`.
3. To reject with the final error, get rid of the `rejectValue` variable and all its associated code. In the `error` function, if the number of call attempts is up, reject the deferred with the value received. Similarly, to return all the errors, keep a list of them and call `reject` with the list once the call limit is reached.
4. The stack never has more than one call to the function passed to `retryingCaller`. A repeat call to that function is not made until the previous call has completely finished (and has returned an error). The stack has therefore been popped before a repeat call is made.
5. You hopefully won’t need any help with this. Arrange to pass an error-checking function and call it in the error callback.

Memoization

1. This is easy, assuming you know your JavaScript. Pass a `context` argument to `memoize` and use `apply` to invoke the passed function.
2. If `func` is called with an object, JavaScript will convert it to a string when accessing `promises[arg]`. This is not very useful because converting an object into a string gets you `'[object Object]'`. So the memoization would fail horribly—all calls that passed an object would pull out the same memoized value, no matter what contents

the object had. The situation is a teeny tiny bit better for arrays, but only if the array is shallow and doesn't contain other arrays or objects.

One way to fix this is to check (in the returned function) the argument type and use something like `JSON.stringify` to turn arrays and objects into strings.

3. If `func` took two arguments, you'd modify `memoize` to return a function that accepted two arguments. It could make the two arguments into a single string for lookup in the `promises` object (note previous challenge).

To generalize this, you're going to need to loop in some way over the passed arguments. An obvious approach would be to build up a lookup string (for accessing `promises`) argument by argument, checking the type of each. There are other ways to skin this cat, though, so we suggest you take a look around on the Web and read some of the many [interesting articles](#).

4. No, this is not a problem. See the summary of "[Deferred Dynamics](#)" on [page 18](#) to understand why.
5. You would need to examine the result of calling `func(arg)`. If it appeared to be a promise, you could store it in `promises` and return it. If not, you'd also want to store the result (perhaps the `promises` variable could be renamed to `results`) and return it. In other words, don't unconditionally call `$.when`.
6. If you didn't wrap the `func(arg)` call in `$.when`, the `promises` object would obviously contain whatever `func(arg)` returned. That could be a promise, but it could be any other value. That's not a problem, except you'd need to tell the caller of `memoize` that they should check the type of the return value, if they care. This is true memoization, rather than the version we wrote initially that always stores (and returns) promises.
7. You can write this one yourself. You should store JavaScript objects in the `promises` object (which you might rename to `promiseInfo`). Each object would contain a promise and a timestamp. You'd update the timestamp on any call for a previously seen argument.

There are two issues to consider here. First, what will happen to the `setInterval` timer once a `memoization` instance goes out of scope and is garbage-collected? Second, what happens if you remove an old item from `promiseInfo` and it contains a promise that has not yet fired?

Short-Term Memoization of In-Progress Function Calls

1. If the promise is added to the `promises` object after the call to `always` (which removes the promise from `promises`), there is the possibility that `internalCreateUser` might finish immediately, in which case the `delete` would fail and the

promise would then be added to `promises`. That would be very unfortunate! An out-of-date promise would be permanently stored in the `promises` object. You might think that this could never happen, but imagine if someday someone speeds up `internalCreateUser` (perhaps *it* is memoized or otherwise cached, or perhaps a test suite stubs it out).

2. The problem above is a good example of a general principle you should follow when maintaining deferreds or promises in data structures. First put the deferred into the data structure you're using to hold them, and only then add callbacks to it to remove it from that data structure when it fires. Never assume that a promise you have received has not already fired.
3. The two are very similar. In both cases, a known result is stored for a finite time and then discarded. With short-term in-progress function call memoization, the result is (effectively) stored in the deferred during the time it takes for the function to complete. With cache expiry (e.g., after a fixed time or on an LRU basis), results are also kept temporarily and then discarded.

Streaming Promise Events

1. The only difference is that another deferred would be created, by `then`. While `then(stream, stream, stream)` looks cute, we don't need to modify and pass on the value delivered to the `notify` method of our deferred, so there's no need to use `then`.
2. Here's how we'd do it. The only point to be careful about is that we should notify the promise about the final resolve or reject of the passed promises before resolving our own deferred:

```
function eventStream(promises){
  var i, deferred = $.Deferred(),
      firedCount = 0,
      callNotify = function(index, type){
        return function(){
          deferred.notify({
            index: index,
            type: type,
            args: [].slice.call(arguments)
          });
        };
      };

  if (type === 'resolve' || type === 'reject'){
    if (++firedCount === promises.length){
      // All promised have fired.
      deferred.resolve();
    }
  }
};
```

```

    });

    for (i = 0; i < promises.length; i++){
        promises[i]
            .done(callNotify(i, 'resolve'))
            .fail(callNotify(i, 'reject'))
            .progress(callNotify(i, 'notify'));
    }

    return deferred.promise();
}

```

3. You can clone a promise using `delegateEventStream` by passing it a handler that just calls the corresponding method on the deferred it is passed:

```

function clonePromise(promise){
    var handler = function(index, type, args, deferred){
        var func;
        if (type === 'resolve'){
            func = deferred.resolve;
        }
        else if (type === 'reject'){
            func = deferred.reject;
        }
        else if (type === 'notify'){
            func = deferred.notify;
        }
        func.apply(deferred, args);
    };

    return delegateEventStream([promise], handler);
}

```

In case you don't get the flow of events here, `delegateEventStream` creates a new promise and returns it, and it's that promise that `clonePromise` also returns. When the original promise (passed to `clonePromise` and passed on to `delegateEventStream`) is resolved or rejected or makes progress, `delegateEventStream` will call the handler defined in `clonePromise` with the information about what happened. To effectively clone the original promise, we just make the same thing happen to the new promise (whose deferred has been passed to our handler by `delegateEventStream`). This is admittedly convoluted, but see if you can think it through. It's good advanced training for thinking about hooking up multiple deferreds.

Getting the First Result from a Set of Promises

1. By now you shouldn't need to pause for even a second to solve this. Use `fail` to add error processing. Remember that your `fail` callback will receive two arguments: the index of the promise that was rejected, and an error value.

2. A no-argument call to `fastestPromise` will result in a promise, as usual. Unfortunately, the promise will never fire! You might think there's no way this could happen—who would be silly enough to call `fastestPromise` with no arguments when you're clearly supposed to give it arguments? But, a call via `fastestPromise.apply` could do it by accidentally supplying an empty argument list. So it's better to be defensive and to throw some kind of error (or, perhaps, to return an already rejected promise holding some kind of error).
3. You could write something like this:

```
var lookup = fastestPromiseWithIndex([
  emailFromRedis('joe'),
  emailFromDatabase('joe')
])
.then(function(index, email){
  var avatarLookup;
  if (index !== 0){
    // Add email address to Redis so we have it cached.
  }
  avatarLookup = fastestPromiseWithIndex([
    avatarFromRedis('joe'),
    avatarFromFilesystem('joe'),
    avatarFromGravatar(email)
  ]);

  avatarLookup.then(function(index, avatar){
    if (index !== 0){
      // The response was not from Redis.
      // Add avatar info to Redis so we have it cached.
    }

    if (index !== 2){
      // The response was not from Gravatar.
      // Cancel the outstanding Gravatar network request.
    }

    return avatar;
  });

  return avatarLookup.promise();
});
```

This introduces a slowdown, though, as all requests to look up an avatar now start with a lookup of the user's email address (to pass to Gravatar). This may be acceptable (due to the Redis caching of email addresses), but in practice you might want to try the Redis and database avatar lookups first. Only if they fail should you look up the email address and make the call to Gravatar.

Note that in a real-world scenario you would need to build this functionality using `delegateEventStream`, not `fastestPromiseWithIndex`, as explained in “[delegateEventStream Redux](#)” on page 59.

4. The following will work, with the usual caveat that testing whether an object is a promise is not bulletproof:

```
function fastestPromiseWithIndex(promiseInfo){
  var item, deferred = $.Deferred(),
      makeDone = function(index){
        return function(result){
          deferred.resolve(index, result);
        };
      },
      makeFail = function(index){
        return function(error){
          deferred.reject(index, error);
        };
      };

  for (var i = 0; i < promiseInfo.length; i++){
    item = promiseInfo[i];
    if (item.promise && jQuery.isFunction(item.promise)){
      item.promise().done(makeDone(i)).fail(makeFail(i));
    }
    else {
      deferred.resolve(i, item);
      break;
    }
  }

  return deferred.promise();
}
```

5. In `eventHandler` where we check if `type === 'reject'`, examine `args`. If a transient network failure is indicated, don't call `deferred.reject`.
6. Exactly as above, handle this when promises are rejected. Differentiate between the different reasons for the rejection.
7. In `eventHandler`, whenever a promise is resolved, we need to check if the status is a 404 and, if so, whether the index is 0 or 1. In either of those cases, arrange to add the avatar to the cache that missed. We can only populate the caches once we actually have the avatar, but we won't yet have it when the fast cache lookups fail. If you've been paying attention, a solution should spring to mind! Just add a `done` callback to the deferred's promise. When the deferred fires later with the eventual result, we can update either or both of the Redis and filesystem caches, as needed. Try something like this:

```
function eventHandler(index, type, args, deferred){
  if (type === 'resolve'){
```

```

    if (args[0].status === 404){
      if (index === 0){
        // Redis look-up failed.
        deferred.promise().done(function(avatar){
          // Add avatar to Redis cache.
        });
      }
      else if (index === 1){
        // Filesystem look-up failed.
        deferred.promise().done(function(avatar){
          // Add avatar to filesystem cache.
        });
      }
    }
    else {
      deferred.resolve.apply(deferred, args);
    }
  }
  else if (type === 'reject'){
    deferred.reject.apply(deferred, args);
  }
}

```

8. The bug will be triggered if all three lookup functions resolve with a 404 status. In that case, the deferred (created by `delegateEventStream` and passed to `eventHandler`) will never be fired. An easy way to fix this is to write a function, `makeEventHandler`, that returns the `eventHandler` function. `makeEventHandler` can have a private variable that counts the number of resolved deferreds. If all three deferreds resolve with a 404 status, `eventHandler` can reject or resolve the deferred in some way, or set up further processing to try to get a result.

A Deferred Queue

1. A priority queue can naturally be implemented with a **heap data structure**. But don't reinvent the wheel more than strictly necessary, unless you want to learn about heaps by implementing one. There are already many JavaScript priority queue implementations.
2. You're largely on your own on this one, sorry! To help with the thinking, it is useful to return a task identifier from `put`. A task identifier can then be passed to the `delete` and `reprioritize` functions, allowing those elements to be found in the priority queue.

One of us wrote a **Twisted (Python) deferred priority queue** with `delete` and `reprioritize` functions that may be a useful point of reference.

when2: An Improved jQuery.when

1. The promise returned by when2 would never be resolved in the cases where you passed it no promise arguments.
2. There would be no change in the behavior of when2, since it doesn't matter if a deferred is resolved or rejected multiple times. The change would be a slight reduction in code size and a slight increase in run time.
3. You might change the end of when2 to look like this:

```
for (i = 0; i < length; i++){
    if (resolveValues[i] &&
        jQuery.isFunction(resolveValues[i].promise)){
        promiseArgFound = true;
        resolveValues[i].promise()
            .done(doneFunc(i))
            .fail(failFunc(i))
            .progress(progressFunc(i));
    }
    else {
        resolveContexts[i] = window;
        --remaining;
    }
}

if (!promiseArgFound){
    if (resolveOnFirstSuccess && length){
        deferred.resolveWith(window, [0, resolveValues[0]]);
    }
    else {
        deferred.resolveWith(resolveContexts, resolveValues);
    }
}

return deferred.promise();
```

4. When we have only one promise, we can use `resolveOnFirstSuccess` and the first to finish (either the `wait` promise or the promise we want to time out) will send its result to the done callback. But if we have multiple promises that we want the results from (assuming they'll all complete before the timeout), we'll only ever get the result of one of them (the first to resolve).
5. The easiest solution is to take what's currently called when the deferreds make progress and instead call it when the deferreds are resolved. That's easily done: just change the callback-adding code in the for loop at the end of when2 to look like this:

```
resolveValues[i].promise()
    .done(progressFunc(i))
```

```
.done(doneFunc(i))  
.fail(failFunc(i));
```

Note that we call `done` with the progress function first. That way, the progress callbacks attached to the `when2` promise will get notified of all deferreds as they finish, before the master deferred is finally resolved with all values. If we had instead written:

```
.done(doneFunc(i)).done(progressFunc(i))
```

the “progress” completion of the last deferred to resolve would not be reported because the `when2` deferred would have already been resolved by the function returned by `doneFunc(i)`.

Timing Out Promises

1. If we removed the `clearTimeout` call in `deactivatingWait`, nothing would change. The timeout would be triggered, but the deferred would have already been resolved. Rejecting a deferred after it has already been resolved has no effect (see “[Deferred Dynamics](#)” on page 18 if you’re unclear on this). The `clearTimeout` call is only included for good housekeeping, and it doesn’t leave someone reading your code to wonder why it’s not cleared (an explanatory comment would help).
2. If `rejectOnFirstError` were not true (or missing from options), timeout expiry would not cause the `when2` deferred to be rejected. `when2WithTimeout` would be completely broken because it would never time out.

Controlling Your Own Destiny

1. `deferredFromPromise` can be made into a one-liner by passing an initialization argument to `$.Deferred()` as follows:

```
function deferredFromPromise(promise){  
  return $.Deferred(function(deferred){  
    promise  
      .done(deferred.resolve)  
      .fail(deferred.reject)  
      .progress(deferred.notify);  
  });  
}
```

2. Just return the deferred! There’s no need to extract its promise and pass that to `deferredFromPromise`.
3. The code will still work just fine. The deferred returned by `deferredFromPromise` will already be resolved. Because the passed promise has fired, the addition of

callbacks to it in `deferredFromPromise` will cause `deferred` to be resolved or rejected (as appropriate) immediately.

Deactivating a Promise

1. The new promise is returned so the result of calling `deactivate` can be chained. Given that you've just deactivated the promise, though, you're highly unlikely to want to add any more callbacks to it! Returning the promise is therefore of questionable utility.
2. Change `unlessDeactivated` to use `apply`, as follows:

```
function unlessDeactivated(){
  return function(){
    if (deactivated === false){
      func.apply(deferred, [].slice.call(arguments));
    }
  };
}
```

3. There are two obvious ways. You could create a new object and copy the public methods (`done`, `fail`, etc.) from `newPromise` to it, as well as giving it a `deactivated` method. That would work, but it's also not future-proof. If jQuery adds a `deactivate` method to promises, that method would not be accessible from the artificial promise returned by `deactivatablePromise`; only our `deactivate` method could be called.

A better solution would be to change `deactivatablePromise` to return a list of two things: the new promise and a separate `deactivate` function.

4. Here's how you might do it:

```
function controllablePromise(promise){
  var deferred = $.Deferred(),
      deactivated = false,
      unlessDeactivated = function(func){
        return function(){
          if (deactivated === false){
            return func.apply(deferred,
                              [].slice.call(arguments));
          }
        };
      };
  resolve = unlessDeactivated(deferred.resolve),
  resolveWith = unlessDeactivated(deferred.resolveWith),

  reject = unlessDeactivated(deferred.reject),
  rejectWith = unlessDeactivated(deferred.rejectWith),
```



```

        notify = unlessDeactivated(deferred.notify),
        notifyWith = unlessDeactivated(deferred.notifyWith);

    promise.done(resolve).fail(reject).progress(notify);

    deferred.resolve = resolve;
    deferred.resolveWith = resolveWith;

    deferred.reject = reject;
    deferred.rejectWith = rejectWith;

    deferred.notify = notify;
    deferred.notifyWith = notifyWith;

    deferred.deactivate = function(){
        deactivated = true;
        return deferred;
    };

    return deferred;
}

```

See the previous challenge to learn why this approach isn't 100% safe.

Bonus challenge! Why does the function returned by `unlessDeactivated` return the deferred? Why did the version in “[Deactivating a Promise](#)” on page 74 not do so?

The Promises/A+ Specification

An open standard for sound, interoperable JavaScript promise—by implementers, for implementers.¹

A *promise* represents the eventual result of an asynchronous operation. The primary way of interacting with a promise is through its `then` method, which registers callbacks to receive either a promise's eventual value or the reason why the promise cannot be fulfilled.

This specification details the behavior of the `then` method, providing an interoperable base which all Promises/A+ conformant promise implementations can be depended on to provide. As such, the specification should be considered very stable. Although the Promises/A+ organization may occasionally revise this specification with minor backward-compatible changes to address newly-discovered corner cases, we will integrate large or backward-incompatible only after careful consideration, discussion, and testing.

Historically, Promises/A+ clarifies the behavioral clauses of the earlier **Promises/A proposal**, extending it to cover *de facto* behaviors and omitting parts that are underspecified or problematic.

Finally, the core Promises/A+ specification does not deal with how to create, fulfill, or reject promises, choosing instead to focus on providing an interoperable `then` method. Future work in companion specifications may touch on these subjects.

1. This appendix reproduces **the Promises/A+ specification**.

Terminology

1. “promise” is an object or function with a `then` method whose behavior conforms to this specification.
2. “thenable” is an object or function that defines a `then` method.
3. “value” is any legal JavaScript value (including `undefined`, a thenable, or a promise).
4. “exception” is a value that is thrown using the `throw` statement.
5. “reason” is a value that indicates why a promise was rejected.

Requirements

Promise States

A promise must be in one of three states: pending, fulfilled, or rejected.

1. When pending, a promise:
 - a. may transition to either the fulfilled or rejected state.
2. When fulfilled, a promise:
 - a. must not transition to any other state.
 - b. must have a value, which must not change.
3. When rejected, a promise:
 - a. must not transition to any other state.
 - b. must have a reason, which must not change.

Here, “must not change” means immutable identity (i.e., `===`), but does not imply deep immutability.

The `then` Method

A promise must provide a `then` method to access its current or eventual value or reason.

A promise’s `then` method accepts two arguments:

```
promise.then(onFulfilled, onRejected)
```

1. Both `onFulfilled` and `onRejected` are optional arguments:
 - a. If `onFulfilled` is not a function, it must be ignored.
 - b. If `onRejected` is not a function, it must be ignored.

2. If `onFulfilled` is a function,
 - a. It must be called after `promise` is fulfilled, with `promise`'s value as its first argument.
 - b. It must not be called before `promise` is fulfilled.
 - c. It must not be called more than once.
3. If `onRejected` is a function,
 - a. It must be called after `promise` is rejected, with `promise`'s reason as its first argument.
 - b. It must not be called before `promise` is rejected.
 - c. It must not be called more than once.
4. `onFulfilled` or `onRejected` must not be called until the **execution context** stack contains only platform code.²
5. `onFulfilled` and `onRejected` must be called as functions (i.e., with no `this` value³).
6. `then` may be called multiple times on the same promise.
 - a. If/when `promise` is fulfilled, all respective `onFulfilled` callbacks must execute in the order of their originating calls to `then`.
 - b. If/when `promise` is rejected, all respective `onRejected` callbacks must execute in the order of their originating calls to `then`.
7. `then` must return a promise.⁴

```
promise2 = promise1.then(onFulfilled, onRejected);
```

 - a. If either `onFulfilled` or `onRejected` returns a value `x`, run the Promise Resolution Procedure `[[Resolve]](promise2, x)`.
 - b. If either `onFulfilled` or `onRejected` throws an exception `e`, `promise2` must be rejected with `e` as the reason.
 - c. If `onFulfilled` is not a function and `promise1` is fulfilled, `promise2` must be fulfilled with the same value.

2. Here “platform code” means engine, environment, and promise implementation code. In practice, this requirement ensures that `onFulfilled` and `onRejected` execute asynchronously, after the event loop turn in which `then` is called, and with a fresh stack. This can be implemented with either a “macro-task” mechanism such as `setTimeout` or `setImmediate`, or with a “micro-task” mechanism such as `MutationObserver` or `process.nextTick`. Since the promise implementation is considered platform code, it may itself contain a task-scheduling queue or “trampoline” in which the handlers are called.
3. That is, in strict mode `this` will be `undefined` inside of them; in sloppy mode, it will be the global object.
4. Implementations may allow `promise2 === promise1`, provided the implementation meets all requirements. Each implementation should document whether it can produce `promise2 === promise1` and under what conditions.

- d. If `onRejected` is not a function and `promise1` is rejected, `promise2` must be rejected with the same reason.

The Promise Resolution Procedure

The *promise resolution procedure* is an abstract operation taking as input a promise and a value, which we denote as `[[Resolve]](promise, x)`. If `x` is a thenable, it attempts to make `promise` adopt the state of `x`, under the assumption that `x` behaves at least somewhat like a promise. Otherwise, it fulfills `promise` with the value `x`.

This treatment of thenables allows promise implementations to interoperate, as long as they expose a Promises/A+-compliant `then` method. It also allows Promises/A+ implementations to “assimilate” nonconformant implementations with reasonable `then` methods.

To run `[[Resolve]](promise, x)`, perform the following steps:

1. If `promise` and `x` refer to the same object, reject `promise` with a `TypeError` as the reason.
2. If `x` is a promise, adopt its state:⁵
 - a. If `x` is pending, `promise` must remain pending until `x` is fulfilled or rejected.
 - b. If/when `x` is fulfilled, fulfill `promise` with the same value.
 - c. If/when `x` is rejected, reject `promise` with the same reason.
3. Otherwise, if `x` is an object or function,
 - a. Let `then` be `x.then`.⁶
 - b. If retrieving the property `x.then` results in a thrown exception `e`, reject `promise` with `e` as the reason.
 - c. If `then` is a function, call it with `x` as `this`, first argument `resolvePromise`, and second argument `rejectPromise`, where:
 - i. If/when `resolvePromise` is called with a value `y`, run `[[Resolve]](promise, y)`.
 - ii. If/when `rejectPromise` is called with a reason `r`, reject `promise` with `r`.
5. Generally, it will only be known that `x` is a true promise if it comes from the current implementation. This clause allows the use of implementation-specific means to adopt the state of known-conformant promises.
6. This procedure of first storing a reference to `x.then`, then testing that reference, and then calling that reference, avoids multiple accesses to the `x.then` property. Such precautions are important for ensuring consistency in the face of an accessor property, whose value could change between retrievals.

- iii. If both `resolvePromise` and `rejectPromise` are called, or multiple calls to the same argument are made, the first call takes precedence, and any further calls are ignored.
- iv. If calling `then` throws an exception `e`,
 - i. If `resolvePromise` or `rejectPromise` have been called, ignore it.
 - ii. Otherwise, reject promise with `e` as the reason.
- d. If `then` is not a function, fulfill promise with `x`.
- 4. If `x` is not an object or function, fulfill promise with `x`.

If a promise is resolved with a thenable that participates in a circular thenable chain, such that the recursive nature of `[[Resolve]](promise, thenable)` eventually causes `[[Resolve]](promise, thenable)` to be called again, following the above algorithm will lead to infinite recursion.

Implementations are encouraged, but not required, to detect such recursion and reject promise with an informative `TypeError` as the reason.⁷

7. Implementations should *not* set arbitrary limits on the depth of thenable chains, and assume that beyond that arbitrary limit the recursion will be infinite. Only true cycles should lead to a `TypeError`; if an infinite chain of distinct thenables is encountered, recursing forever is the correct behavior.

Converting an ArrayBuffer to Base 64

The `base64ArrayBuffer` function we used in “[Displaying Google Maps](#)” on page 32 was written by Nicolas Perriault and [posted on StackOverflow](#).

The (slightly cleaned-up) code is reproduced below.

```
function base64ArrayBuffer(arrayBuffer){
    var base64    = '',
        encodings = ('ABCDEFGHIJKLMNOPQRSTUVWXYZ' +
                     'abcdefghijklmnopqrstuvwxyz' +
                     '0123456789+/'),

        bytes      = new Uint8Array(arrayBuffer),
        byteLength = bytes.byteLength,
        byteRemainder = byteLength % 3,
        mainLength  = byteLength - byteRemainder,

        a, b, c, d,
        chunk,
        i;

    // Main loop deals with bytes in chunks of 3
    for (i = 0; i < mainLength; i = i + 3){
        // Combine the three bytes into a single integer
        chunk = (bytes[i] << 16) | (bytes[i + 1] << 8) | bytes[i + 2];

        // Use bitmasks to extract 6-bit segments from the triplet
        a = (chunk & 16515072) >> 18; // 16515072 = (2^6 - 1) << 18
        b = (chunk & 258048) >> 12; // 258048 = (2^6 - 1) << 12
        c = (chunk & 4032) >> 6; // 4032 = (2^6 - 1) << 6
        d = chunk & 63; // 63 = 2^6 - 1

        // Convert the raw binary segments to the appropriate ASCII encoding
        base64 += encodings[a] + encodings[b] + encodings[c] + encodings[d];
    }
}
```

```

// Deal with the remaining bytes and padding
if (byteRemainder === 1){
    chunk = bytes[mainLength];

    a = (chunk & 252) >> 2; // 252 = (2^6 - 1) << 2

    // Set the 4 least significant bits to zero
    b = (chunk & 3) << 4; // 3 = 2^2 - 1

    base64 += encodings[a] + encodings[b] + '=';
} else if (byteRemainder === 2){
    chunk = (bytes[mainLength] << 8) | bytes[mainLength + 1];

    a = (chunk & 64512) >> 10; // 64512 = (2^6 - 1) << 10
    b = (chunk & 1008) >> 4; // 1008 = (2^6 - 1) << 4

    // Set the 2 least significant bits to zero
    c = (chunk & 15) << 2; // 15 = 2^4 - 1

    base64 += encodings[a] + encodings[b] + encodings[c] + '=';
}

return base64;
}

```

About the Authors

Terry Jones is the founder and CTO of [Fluidinfo](#), and has been programming for 35 years. He has worked extensively with deferreds for the last 7 years and, strange to say, has become passionate about them. In particular, he loves how elegantly deferreds can be clicked together to build more complex things and how thinking about them has broadened his perspective on programming. There's more about Terry on his out-of-date [web pages](#) and he also [blogs regularly](#). He can be reached at terry@jon.es.

Nicholas H. Tollervey is a classically trained musician, philosophy graduate, teacher, writer, and software developer. He's just like this biography: concise, honest, and full of useful information. He blogs at ntoll.org, tweets as [ntoll](#) and can be reached at ntoll@ntoll.org.

Colophon

The animal on the cover of *Learning jQuery Deferreds* is a musky rat-kangaroo (*Hypsiprymnodon moschatus*).

The cover image is from Johnson's *Natural History*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.